(12) UK Patent Application (19) GB (11) 2 370 658 (13) A

(43) Date of A Publication 03.07.2002

(21) Application No 0031821.2

(22) Date of Filing 29.12.2000

(71) Applicant(s)
Metadyne Limited
(Incorporated in the United Kingdom)
Riverside House, 47 The Lynch, UXBRIDGE, Middlesex, UB8 2TQ, United Kingdom

(72) Inventor(s)
Joseph Timothy Poole
Andrew David Evans

(74) Agent and/or Address for Service
Reddie & Grose
16 Theobalds Road, LONDON, WC1X 8PL, United Kingdom

(51) INT CL^7
G06F 9/44 9/46 9/52

(52) UK CL (Edition T)
G4A APL APX

(56) Documents Cited
GB 2353612 A **GB 2340265 A**
EP 0315493 A2 **US 6154763 A**
Dr Dobb's Journal, April 1997, CD-ROM version, "Java Beans and the New Event Model", Giguere E.

(58) Field of Search
UK CL (Edition S) **G4A APX**
INT CL^7 **G06F 9/44 9/46 9/52**
Dr Dobb's Journal.
Online: JAPIO, EPODOC, WPI, Internet

(54) Abstract Title
A modular software framework

(57) A software platform is described which supports the real time construction and modification of applications comprised of a plurality of software modules. Software modules are implemented to perform the smallest possible role or function within an application so that there is as little as possible overlap between the functionality provided by different modules. As a user of the application, or other software modules of the application, request some functionality, the software modules which provide that functionality are loaded in and connected to the application. Modules which provide functionality that is no longer needed are disconnected from the application so that the application takes up as little space in memory as possible.

In the preferred embodiment the platform is provided by a number of Java (RTM) APIs which extend the functions provided by Sun Microsystem's Java platform. The classes and interfaces of the APIs allow software modules 74, 76, 79 to be classified so that they may be integrated into an application in a known manner, and so that they may be identified easily and requested for use; they also provide a Registry of available software modules that may be used with the application, a data model for representing data in a uniform way within an application and a communication protocol for sharing data and configuration information 72, 78 between software modules.

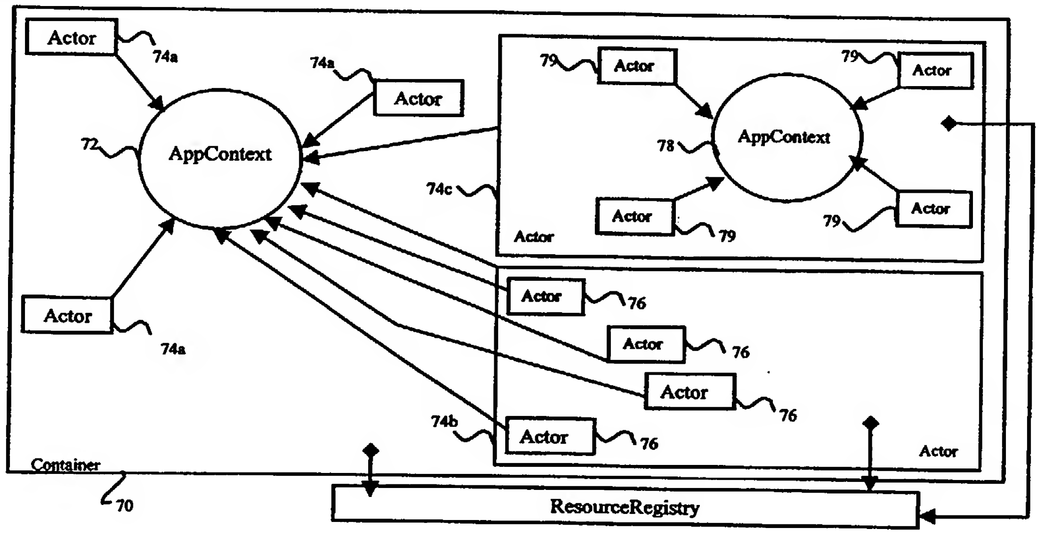


Fig. 7

GB 2 370 658 A

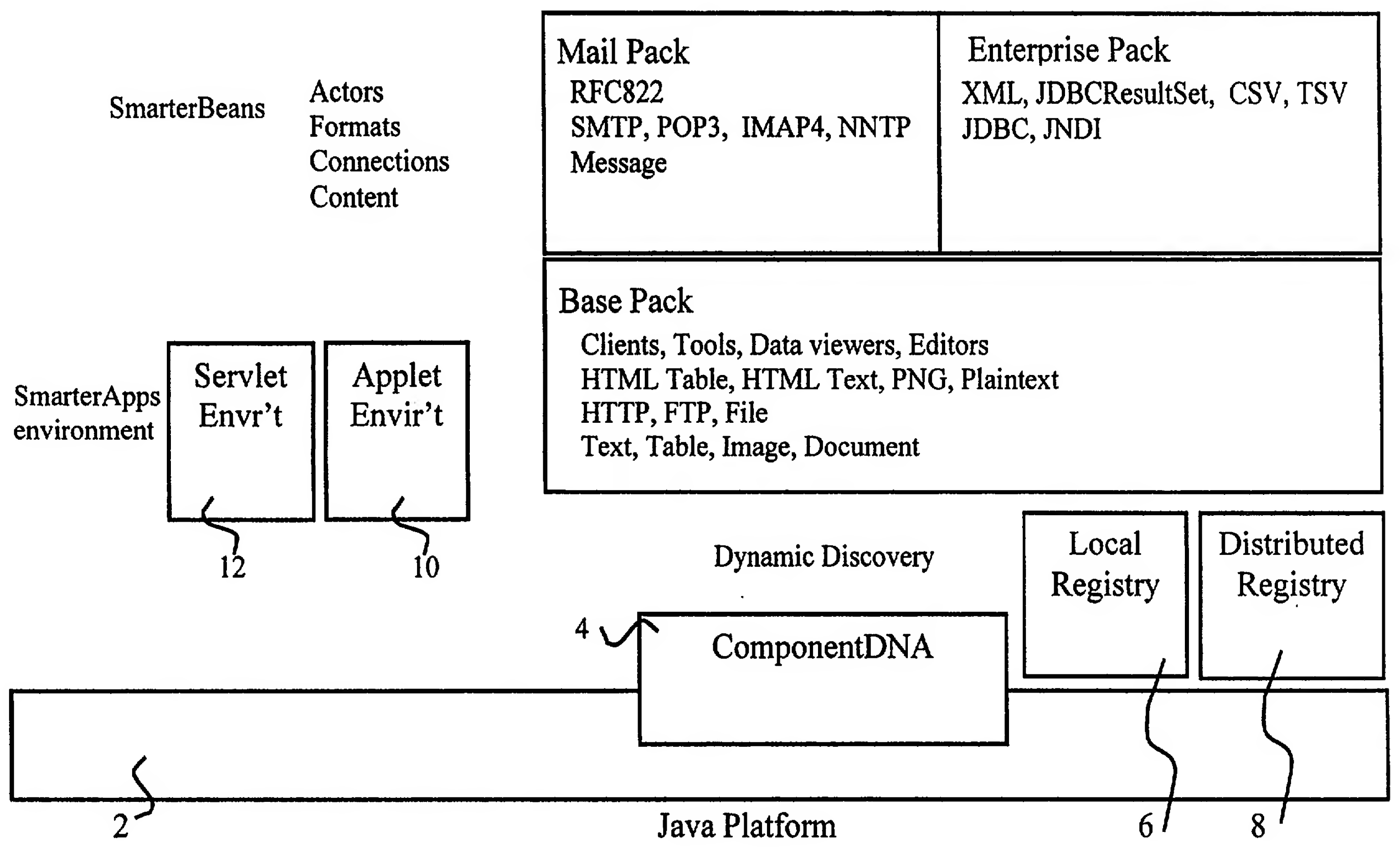

SmarterBeans
Actors
Formats
Connections
Content
Mail Pack
RFC822
SMTP, POP3, IMAP4, NNTP
Message
Enterprise Pack
XML, JDBCResultSet, CSV, TSV
JDBC, JNDI
Base Pack
Clients, Tools, Data viewers, Editors
HTML Table, HTML Text, PNG, Plaintext
HTTP, FTP, File
Text, Table, Image, Document
SmarterApps environment
Servlet Envr't
Applet Envir't
12
10
Dynamic Discovery
Local Registry
Distributed Registry
4
ComponentDNA
2
Java Platform
6
8

Fig. 1

The Resource Registry API

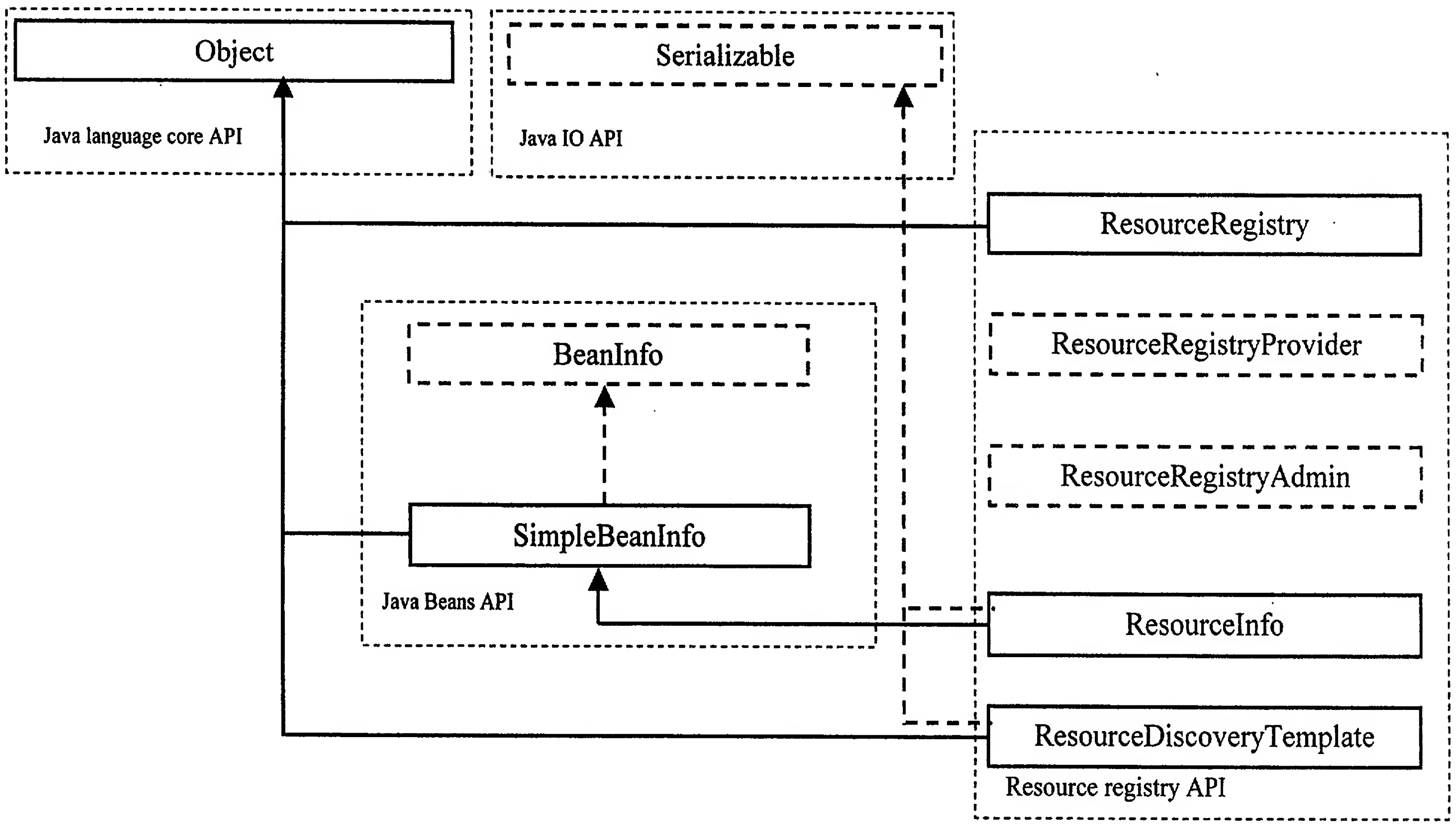
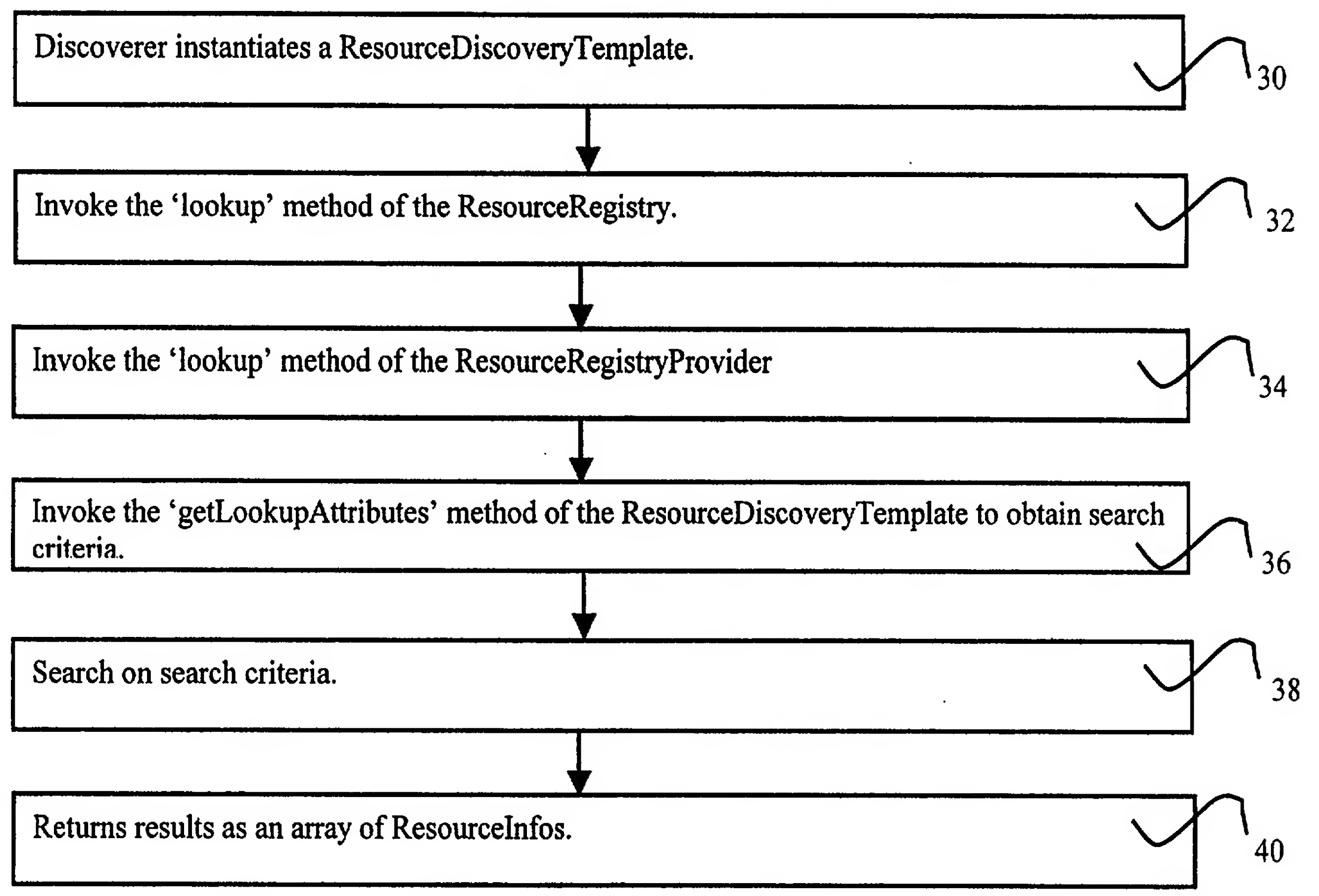


Fig. 2



Discoverer instantiates a ResourceDiscoveryTemplate.
30
Invoke the ‘lookup’ method of the ResourceRegistry.
32
Invoke the ‘lookup’ method of the ResourceRegistryProvider
34
Invoke the ‘getLookupAttributes’ method of the ResourceDiscoveryTemplate to obtain search criteria.
36
Search on search criteria.
38
Returns results as an array of ResourceInfos.
40

Fig. 3

The Actor API

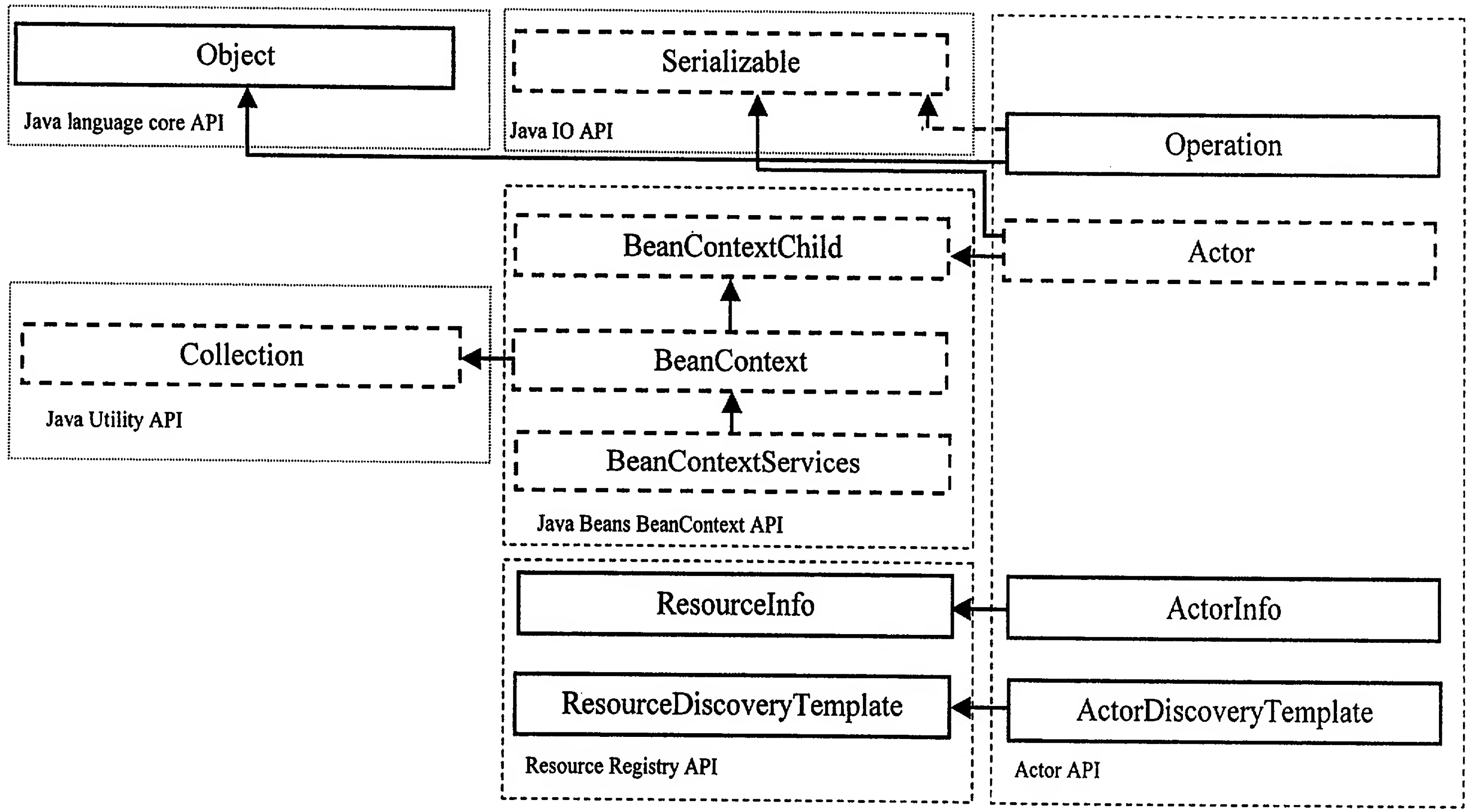


Fig. 4

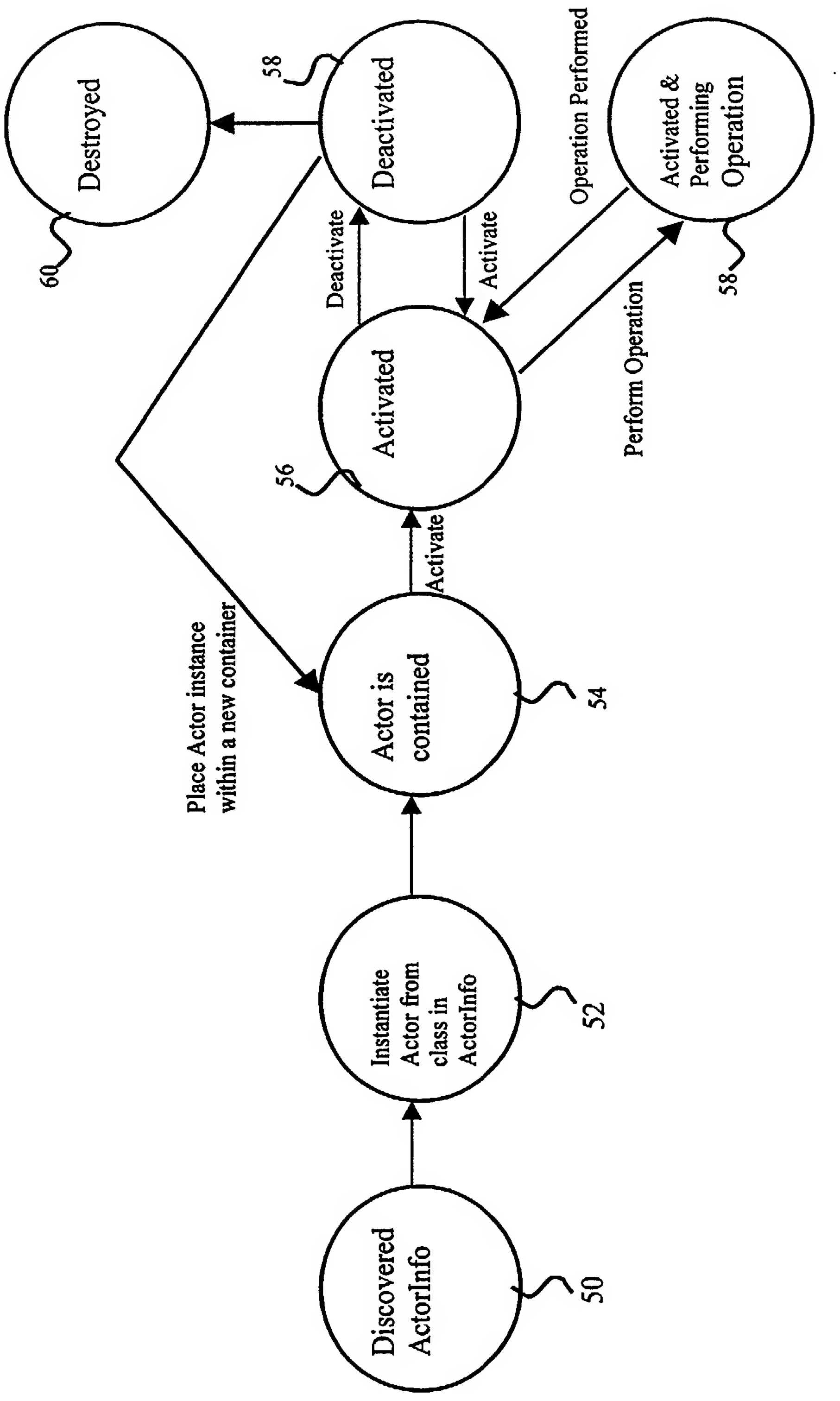

Discovered ActorInfo
50
Instantiate Actor from class in ActorInfo
52
Actor is contained
54
Place Actor instance within a new container
Activate
Activated
56
Deactivate
Activate
Deactivated
58
Destroyed
60
Perform Operation
Operation Performed
Activated & Performing Operation
58

Fig. 5

The AppContext API

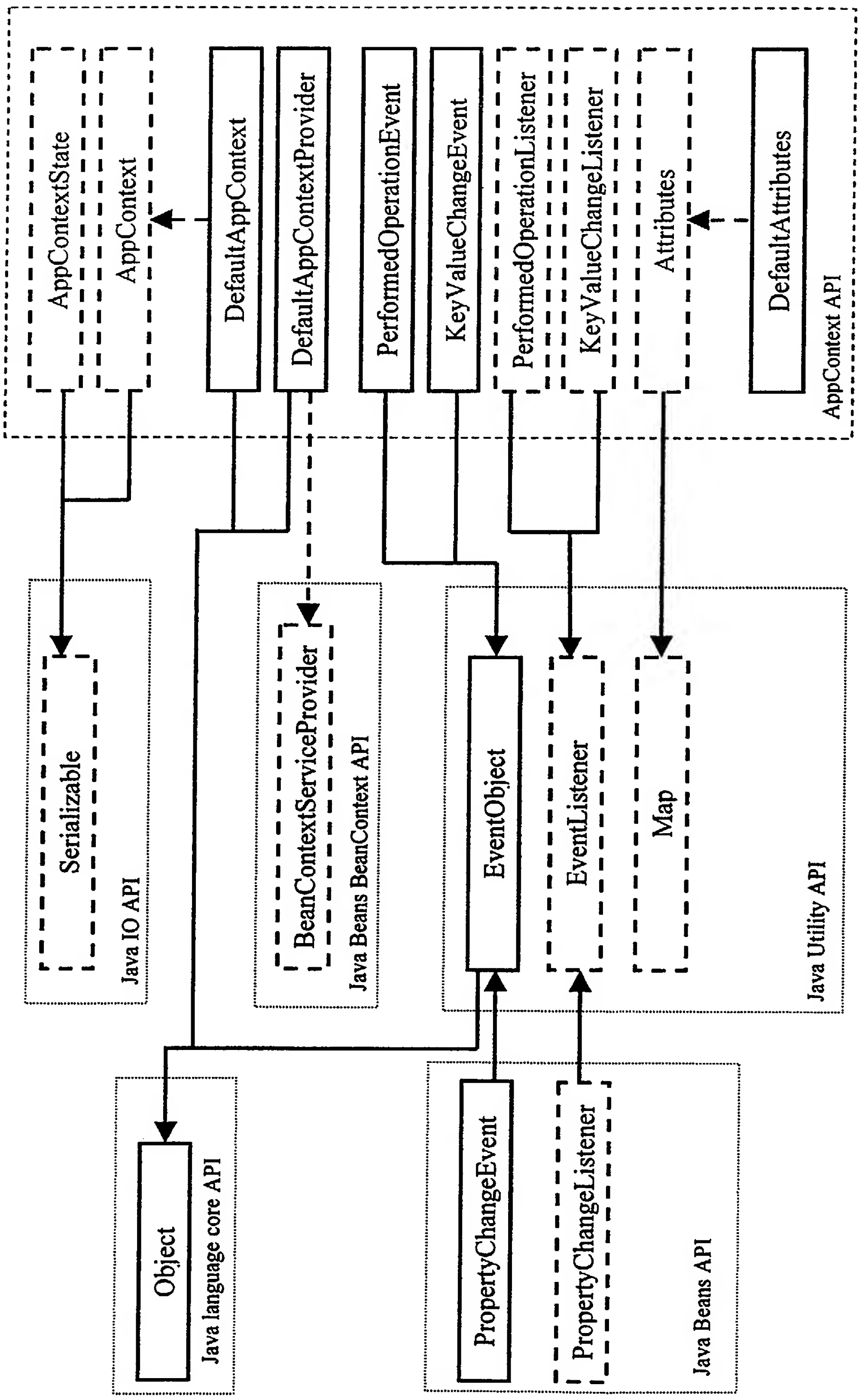


Fig. 6

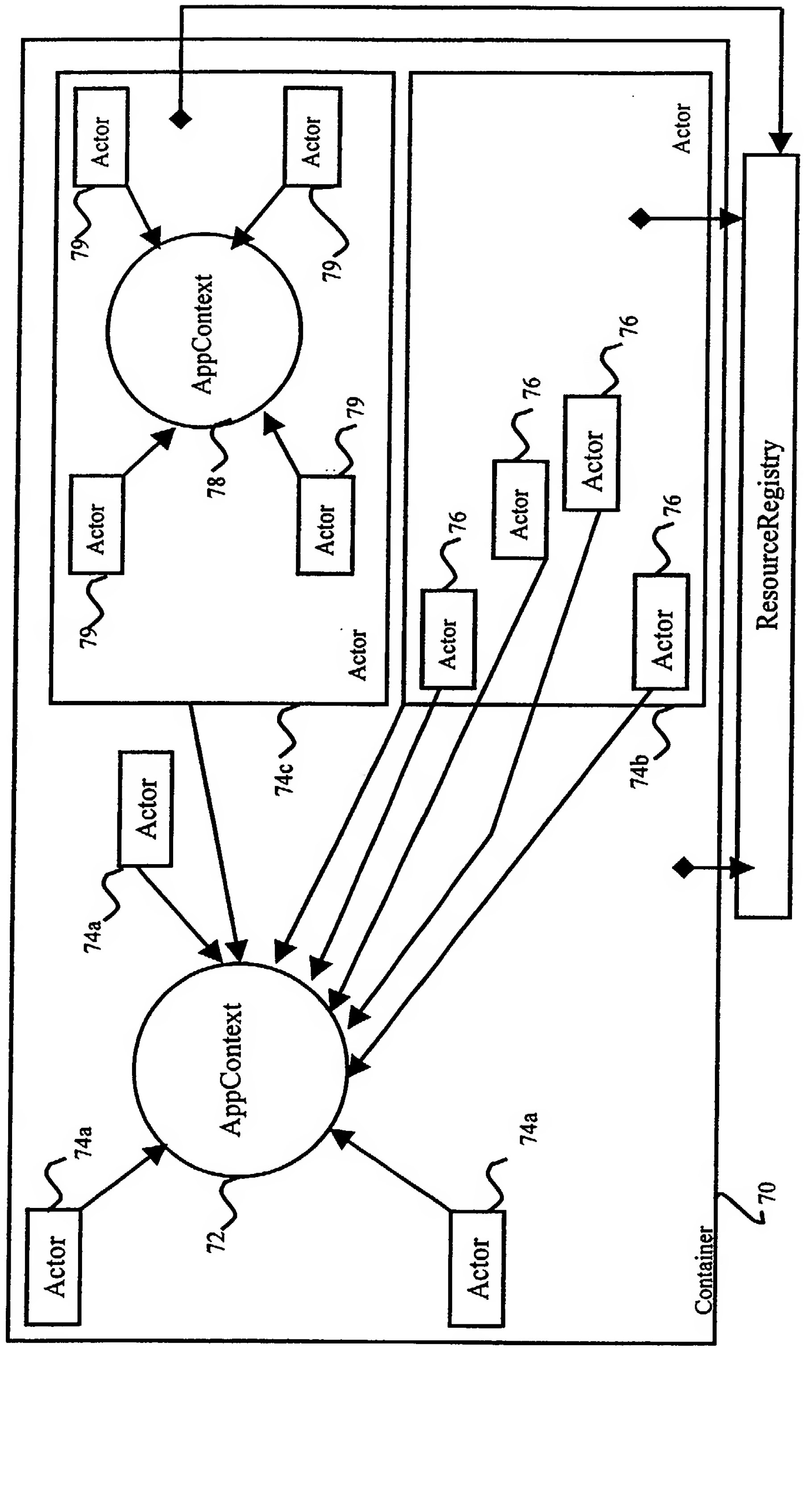

Container
70
AppContext
72
Actor
74a
Actor
74a
Actor
74a
74c
Actor
AppContext
78
Actor
79
Actor
79
Actor
79
Actor
79
74b
Actor
Actor
76
Actor
76
Actor
76
Actor
76
ResourceRegistry

Fig. 7

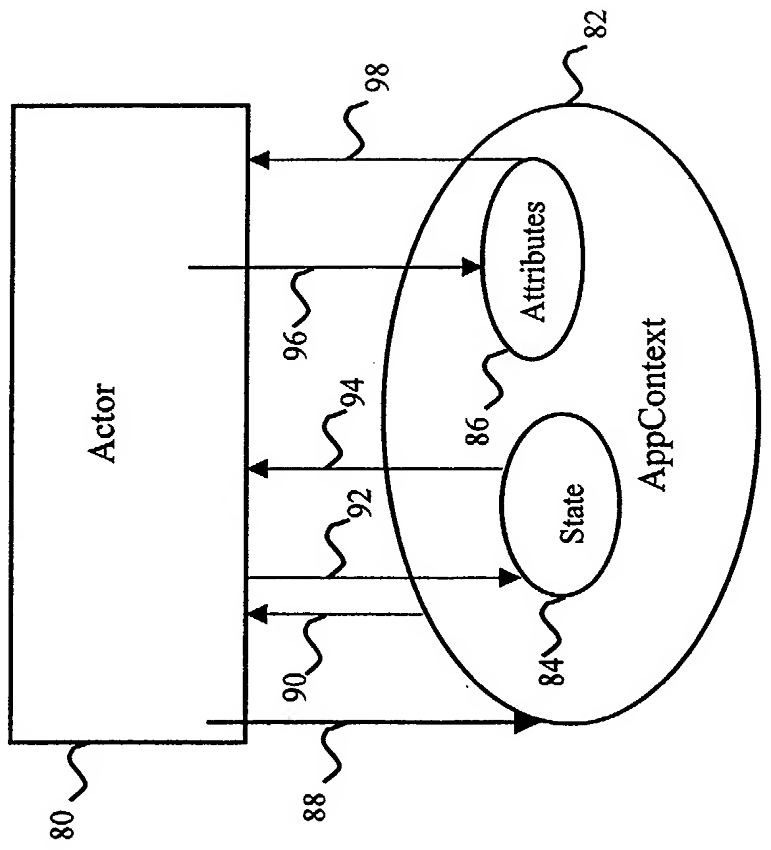

Actor
AppContext
State
Attributes
80
82
84
86
88
90
92
94
96
98

Fig. 8

The Node and Content API

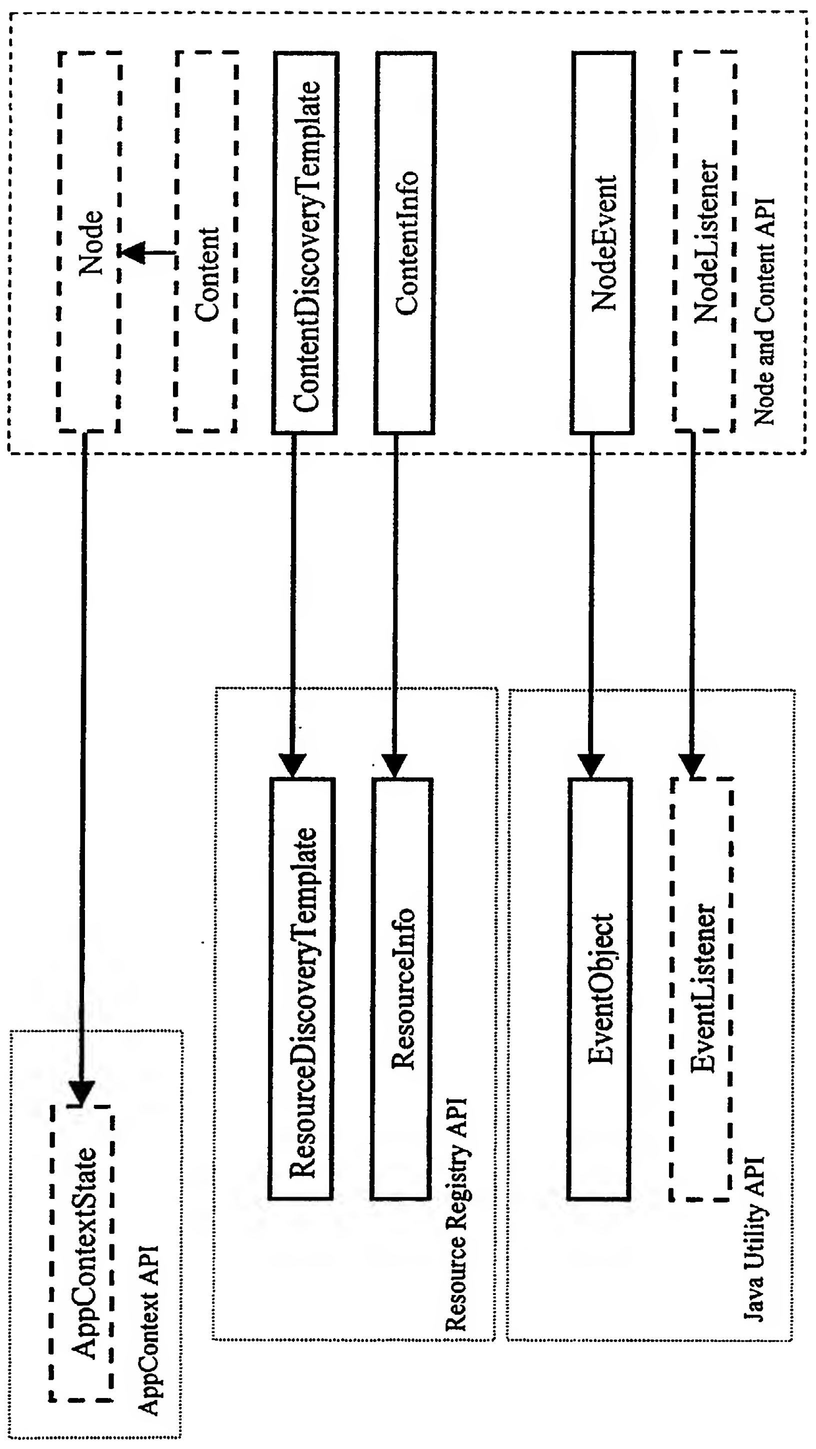


Fig. 9

The Connection API

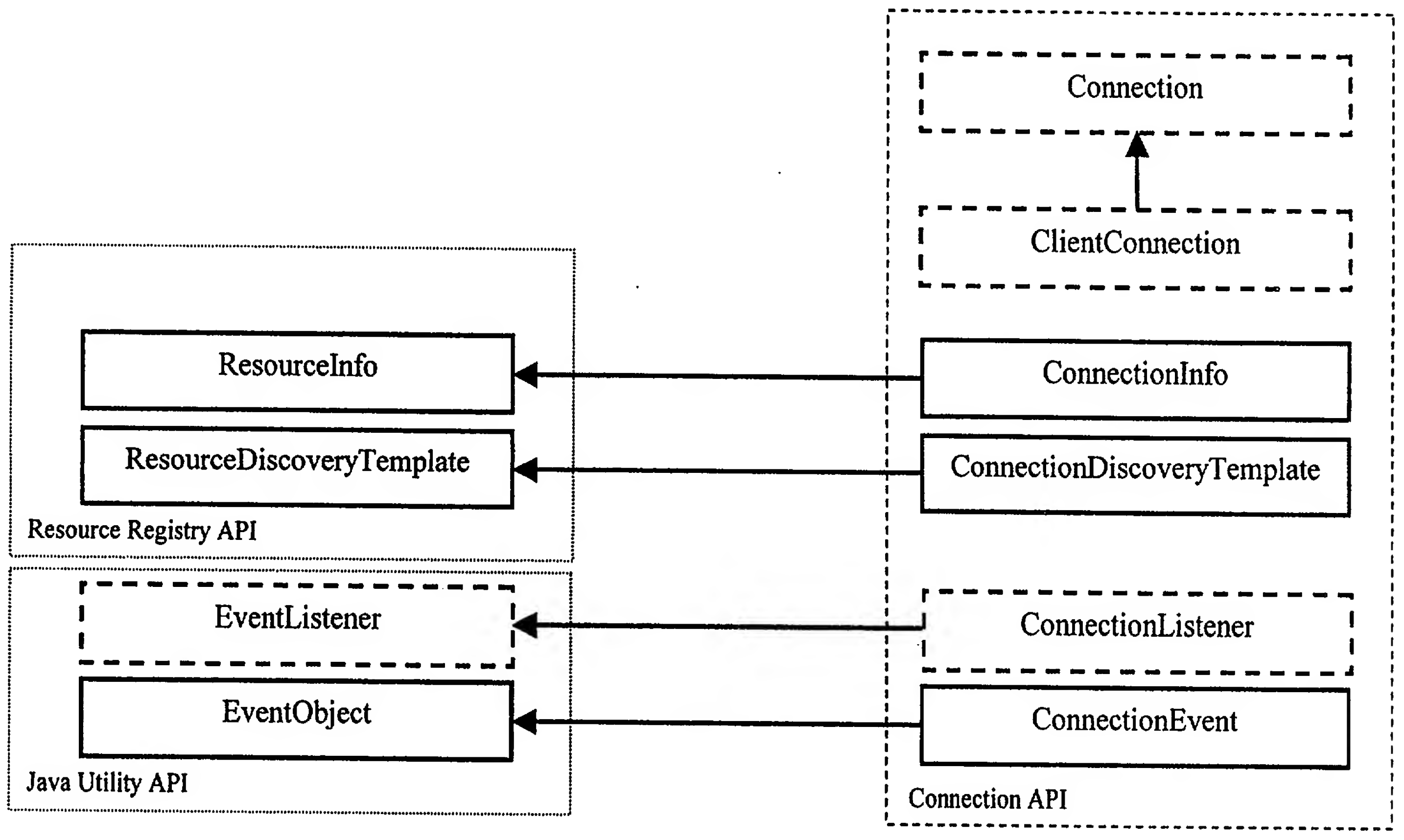


Fig. 10

The Format API

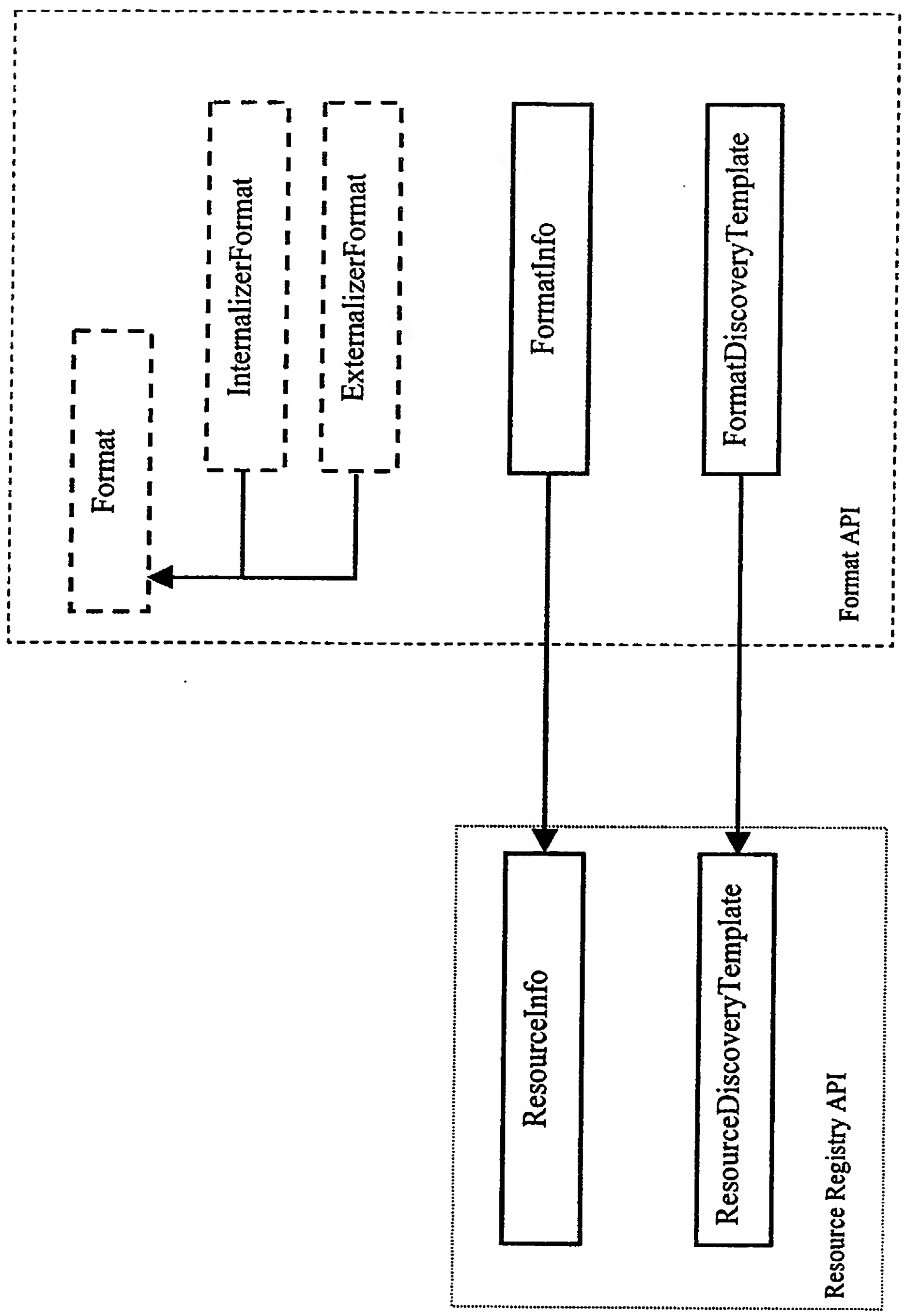


Fig. 11

Example resource implementations

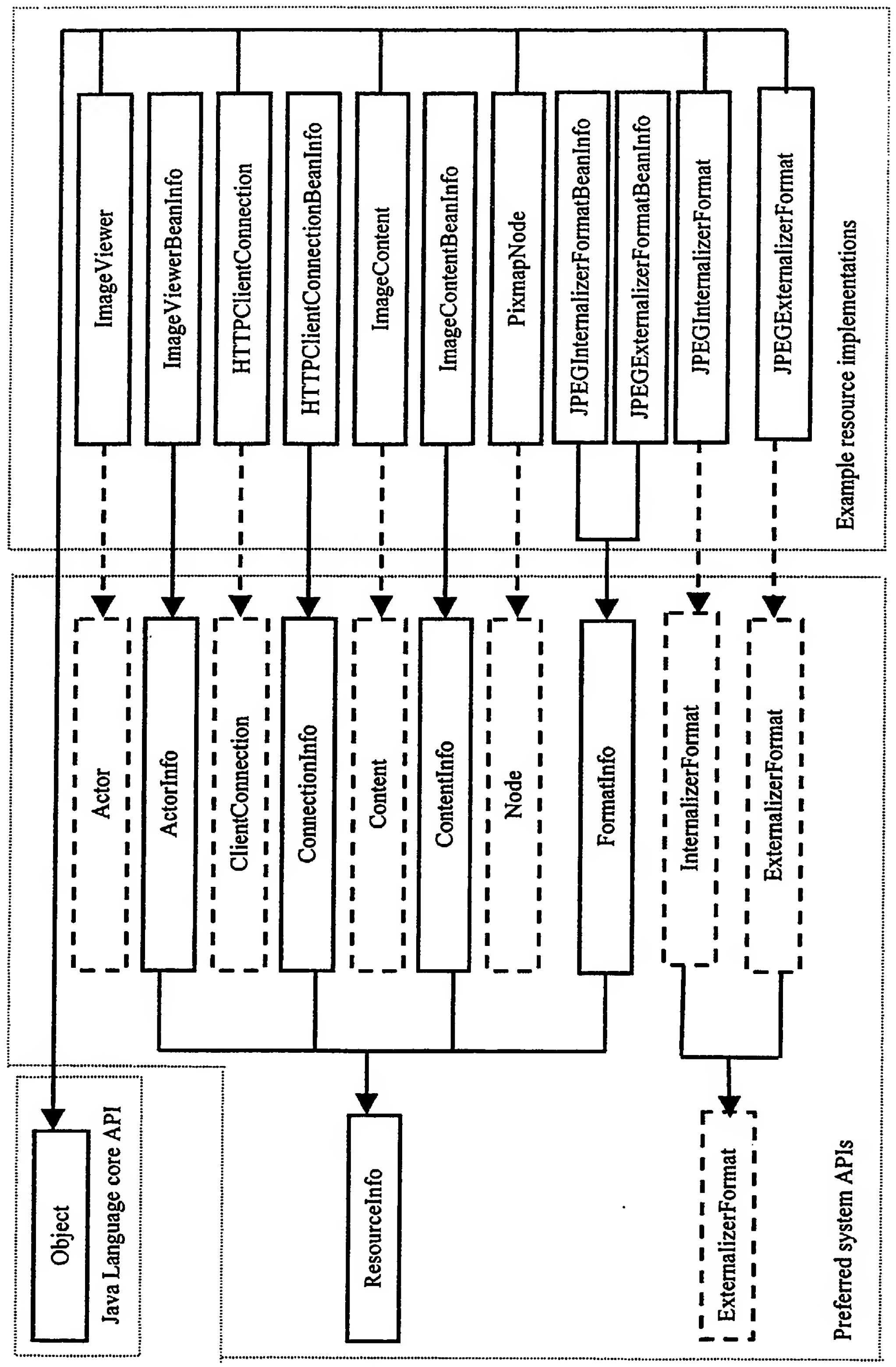


Fig. 12

15 4 5 02

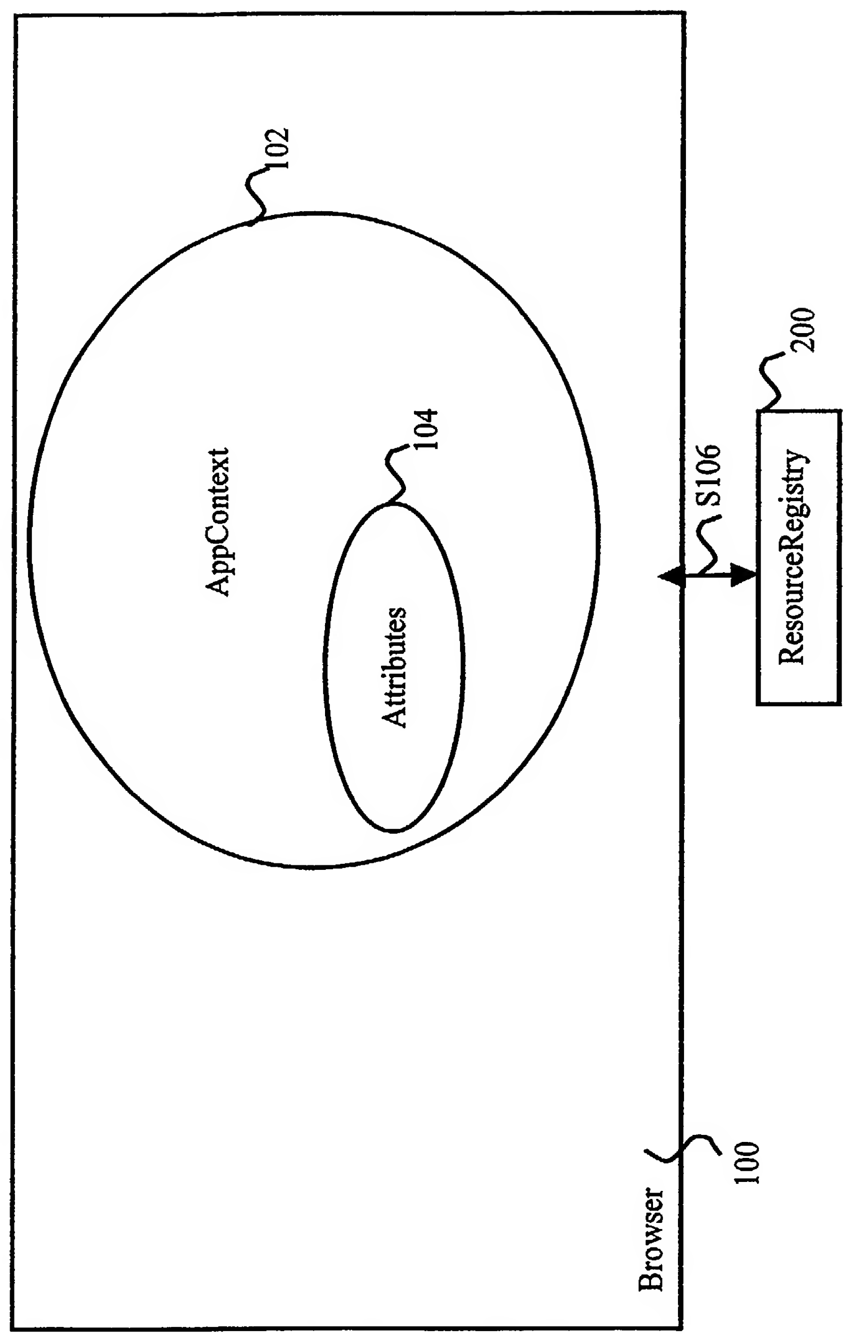
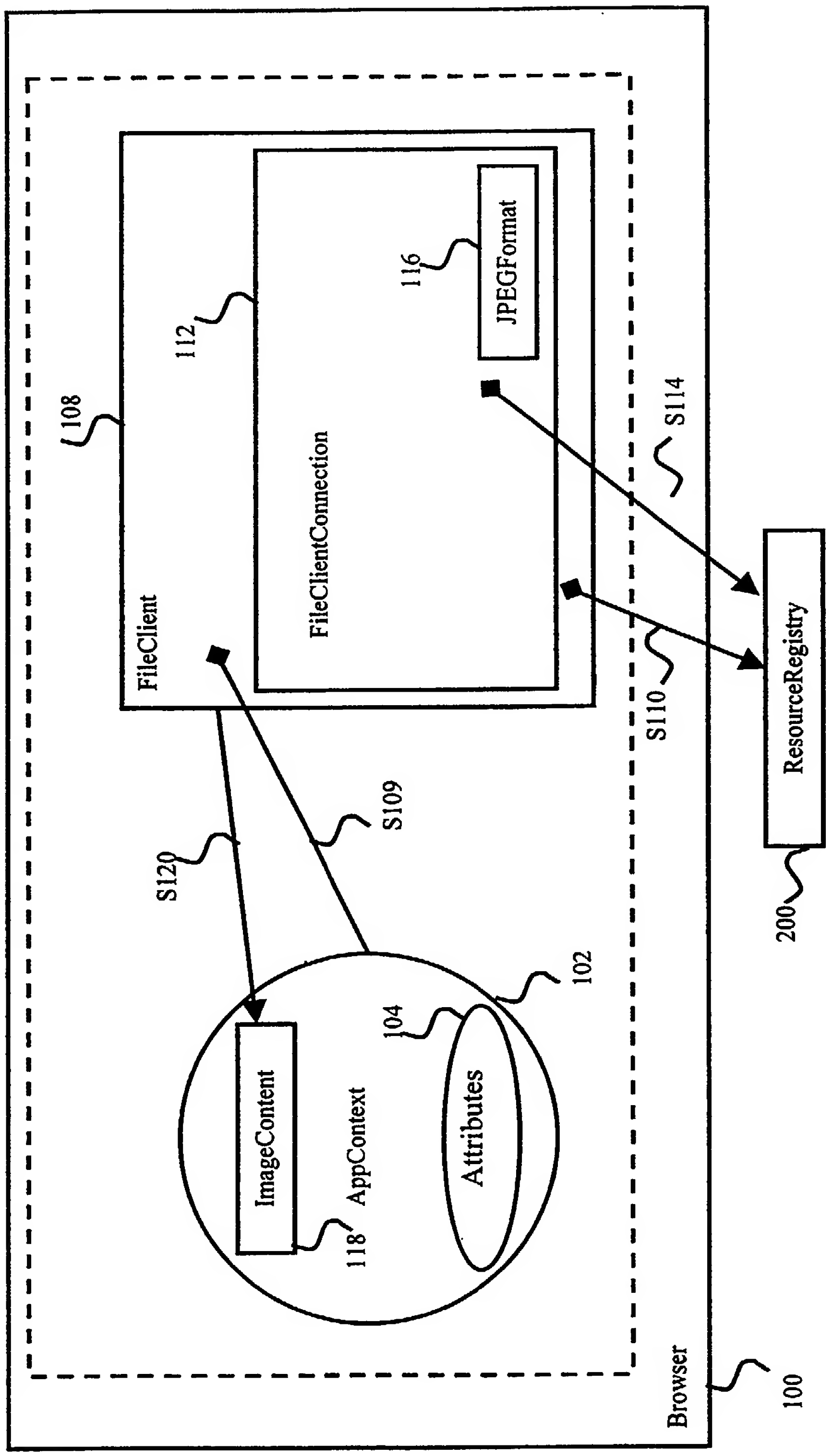


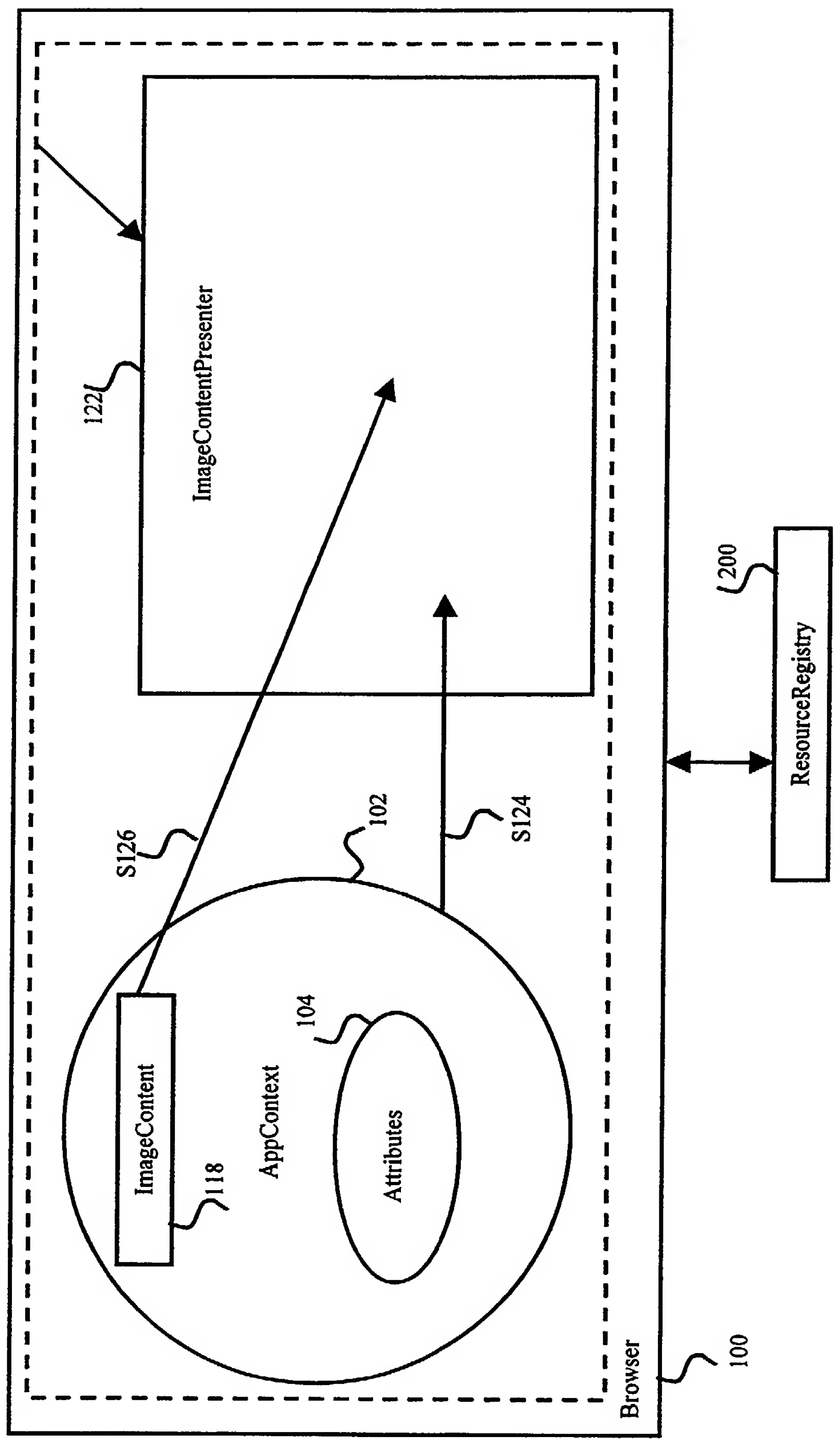
Fig. 13

15 ·5 02



Browser
100
108
FileClient
112
FileClientConnection
116
JPEGFormat
S114
S110
ResourceRegistry
200
S109
S120
102
AppContext
ImageContent
118
Attributes
104

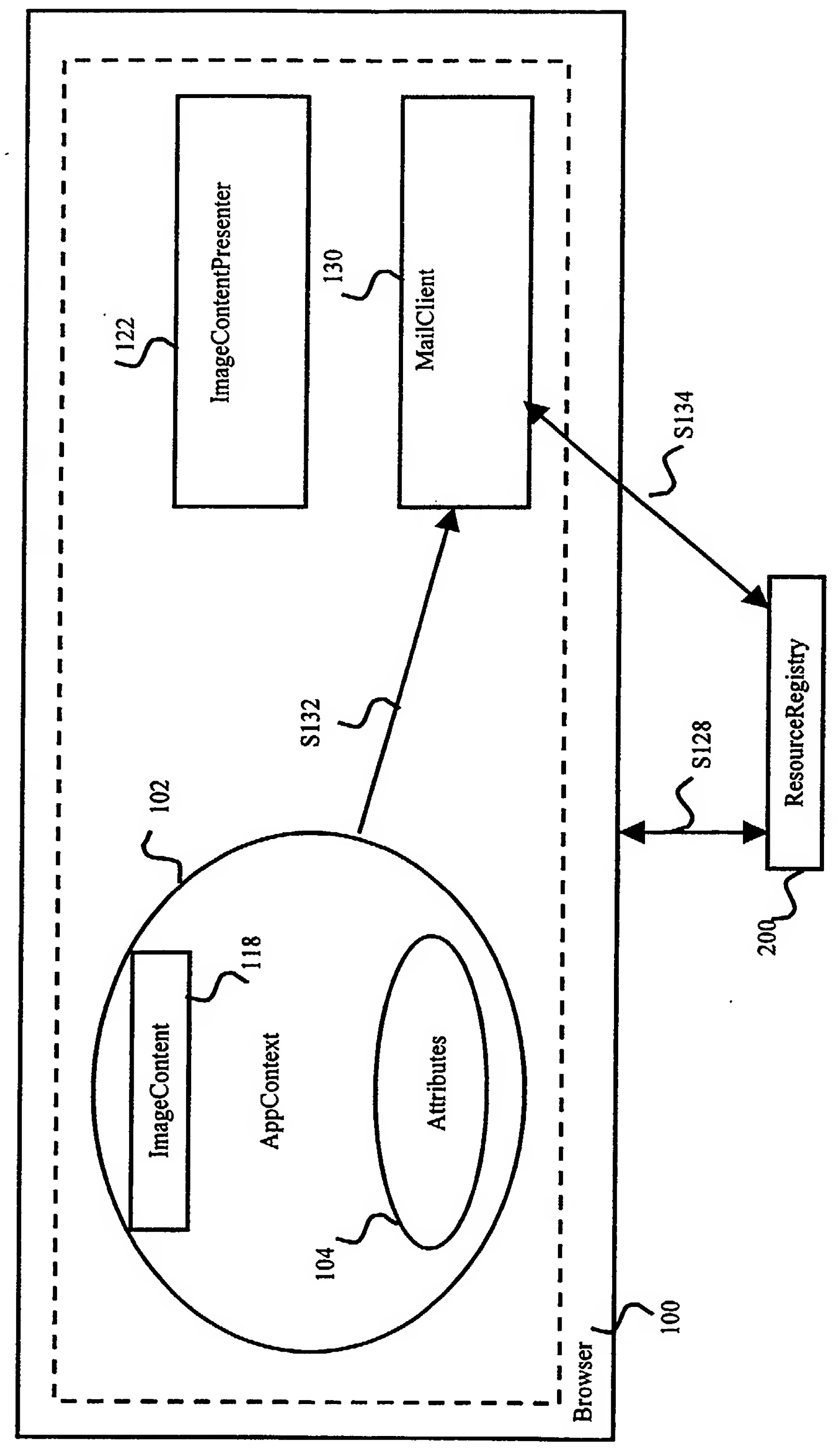
Fig. 14



ImageContentPresenter
122
S126
102
S124
200
ResourceRegistry
ImageContent
AppContext
104
Attributes
118
Browser
100

Fig. 15

15.5.02



ImageContentPresenter
122
130
MailClient
S134
S132
102
S128
ResourceRegistry
200
118
ImageContent
AppContext
Attributes
104
Browser
100

Fig. 16

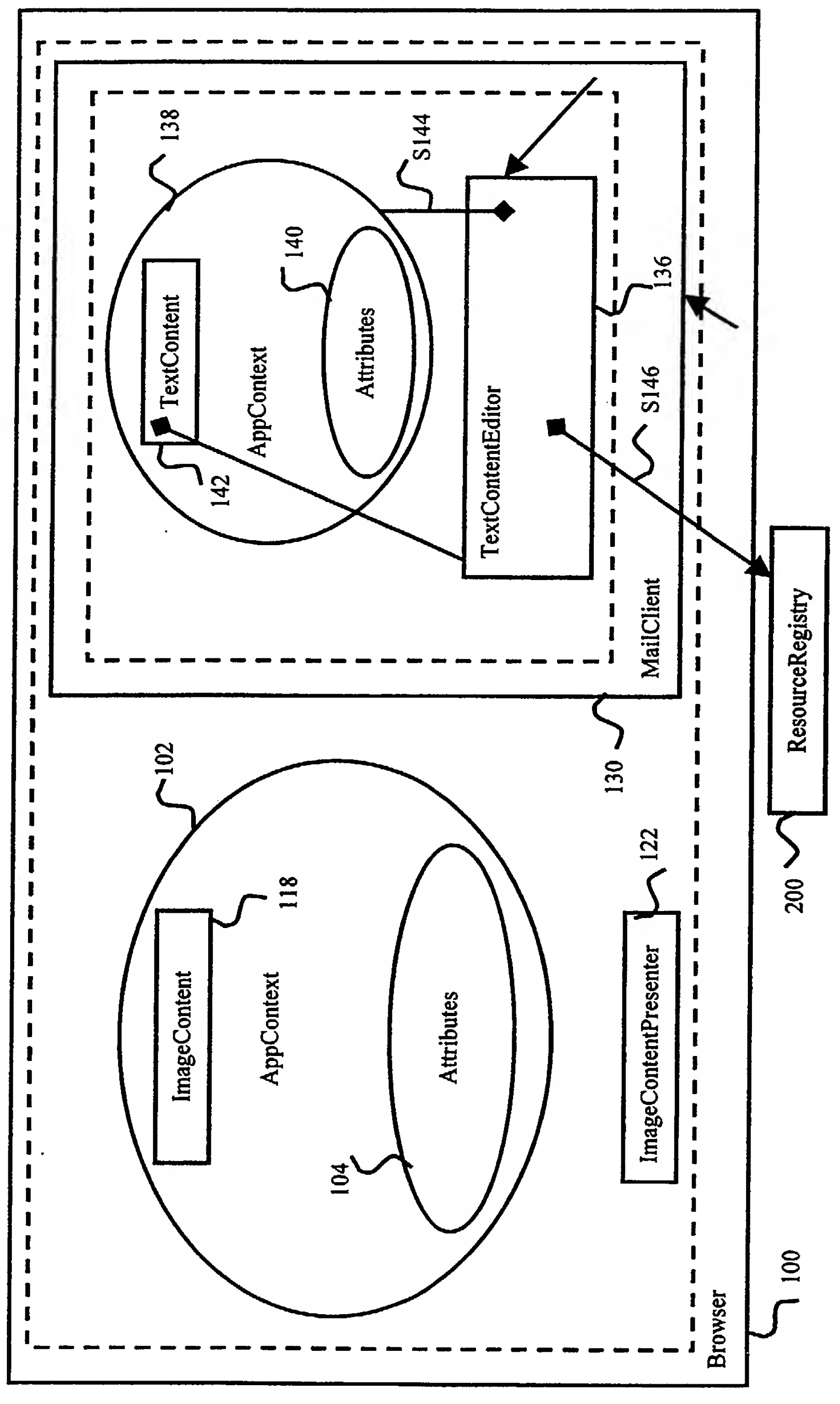
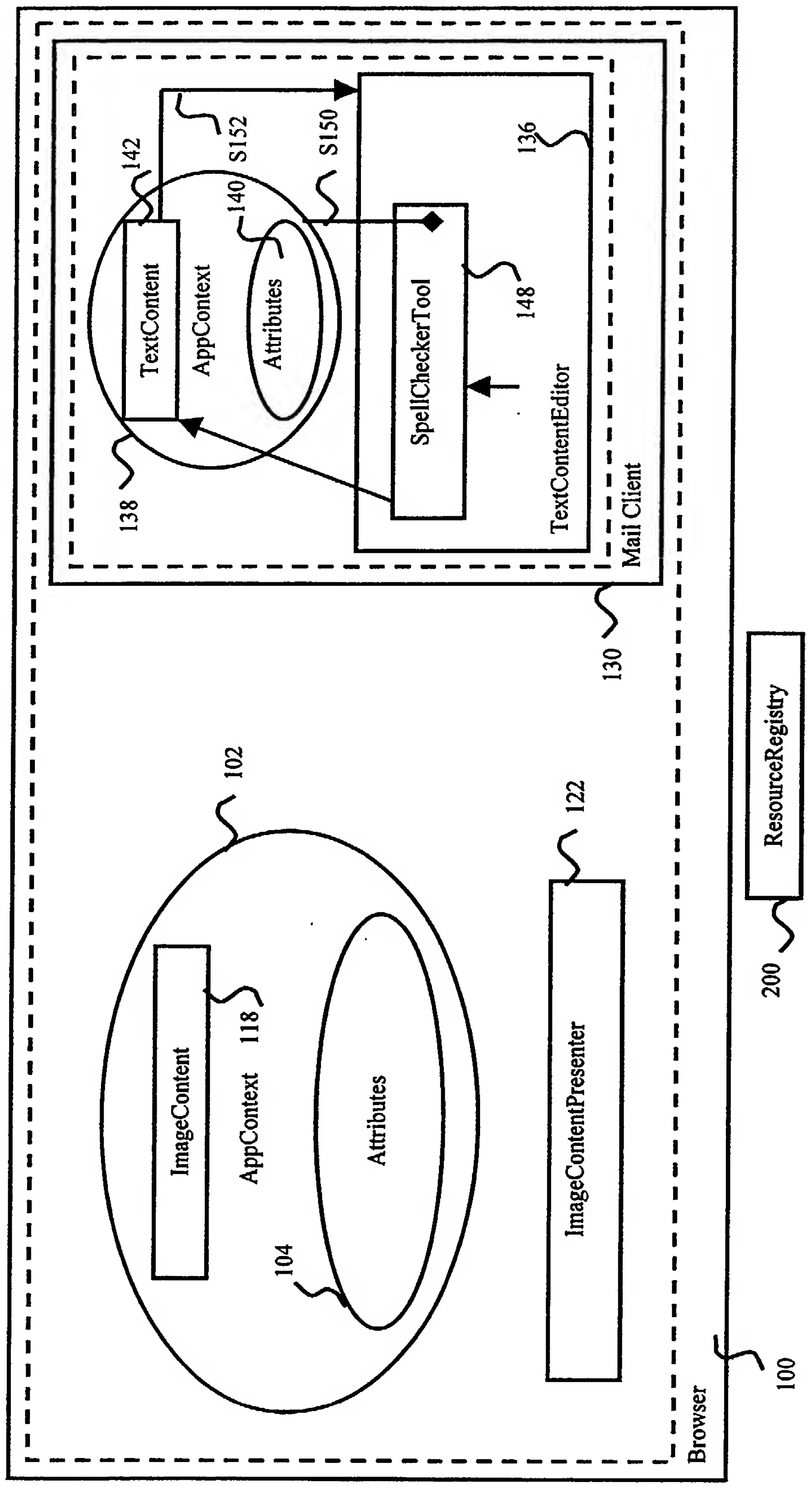

Browser
100
130
MailClient
136
138
AppContext
TextContent
142
140
Attributes
TextContentEditor
S144
S146
ResourceRegistry
200
102
AppContext
ImageContent
118
Attributes
104
ImageContentPresenter
122

Fig. 17

15 5 02



Browser
100
ImageContent
118
AppContext
102
Attributes
104
ImageContentPresenter
122
ResourceRegistry
200
130
Mail Client
TextContentEditor
136
TextContent
142
AppContext
138
Attributes
140
SpellCheckerTool
148
S150
S152

Fig. 18

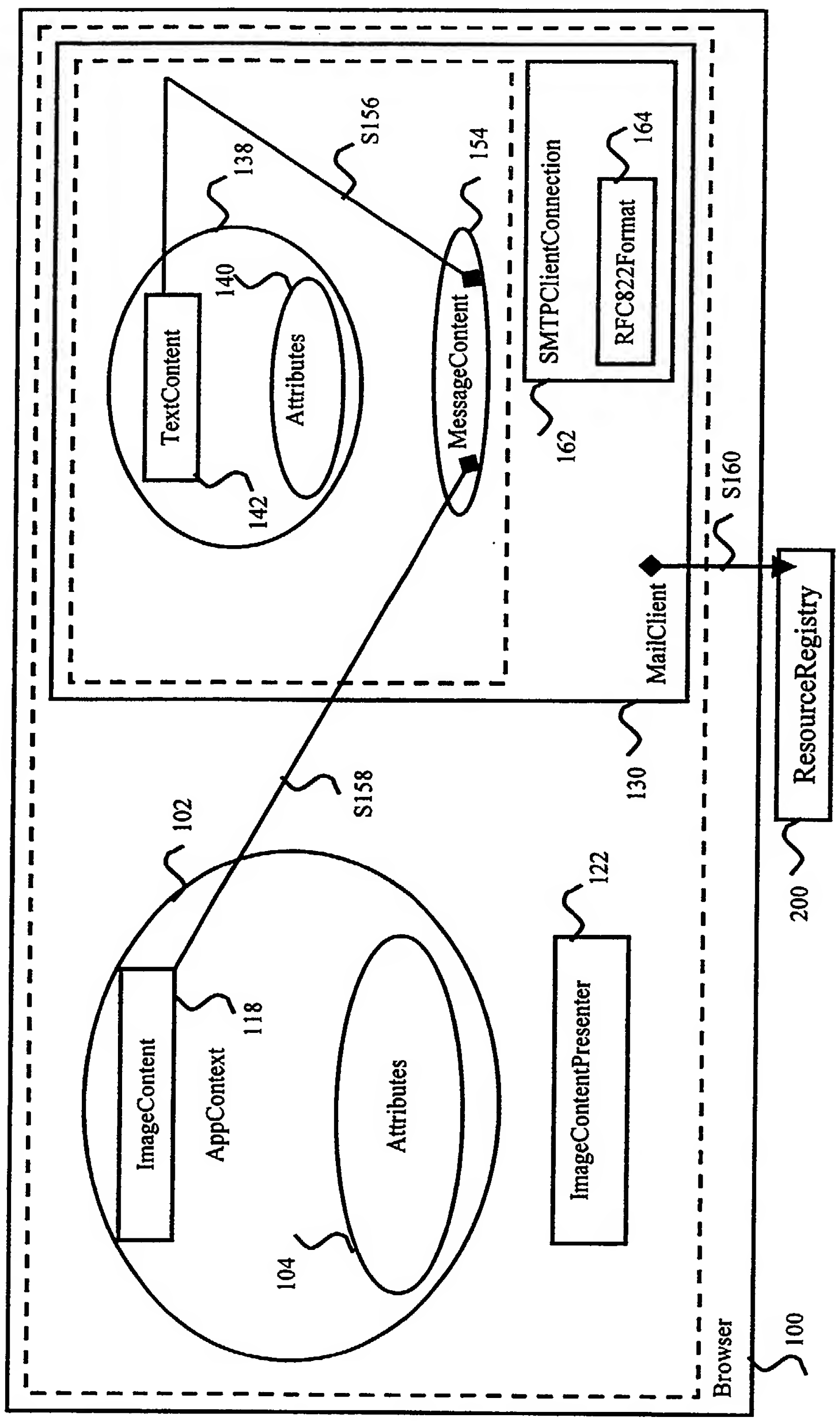

Browser
100
AppContext
ImageContent
118
102
Attributes
104
ImageContentPresenter
122
MailClient
130
TextContent
142
Attributes
140
138
MessageContent
154
S156
S158
SMTPClientConnection
162
RFC822Format
164
S160
ResourceRegistry
200

Fig. 19

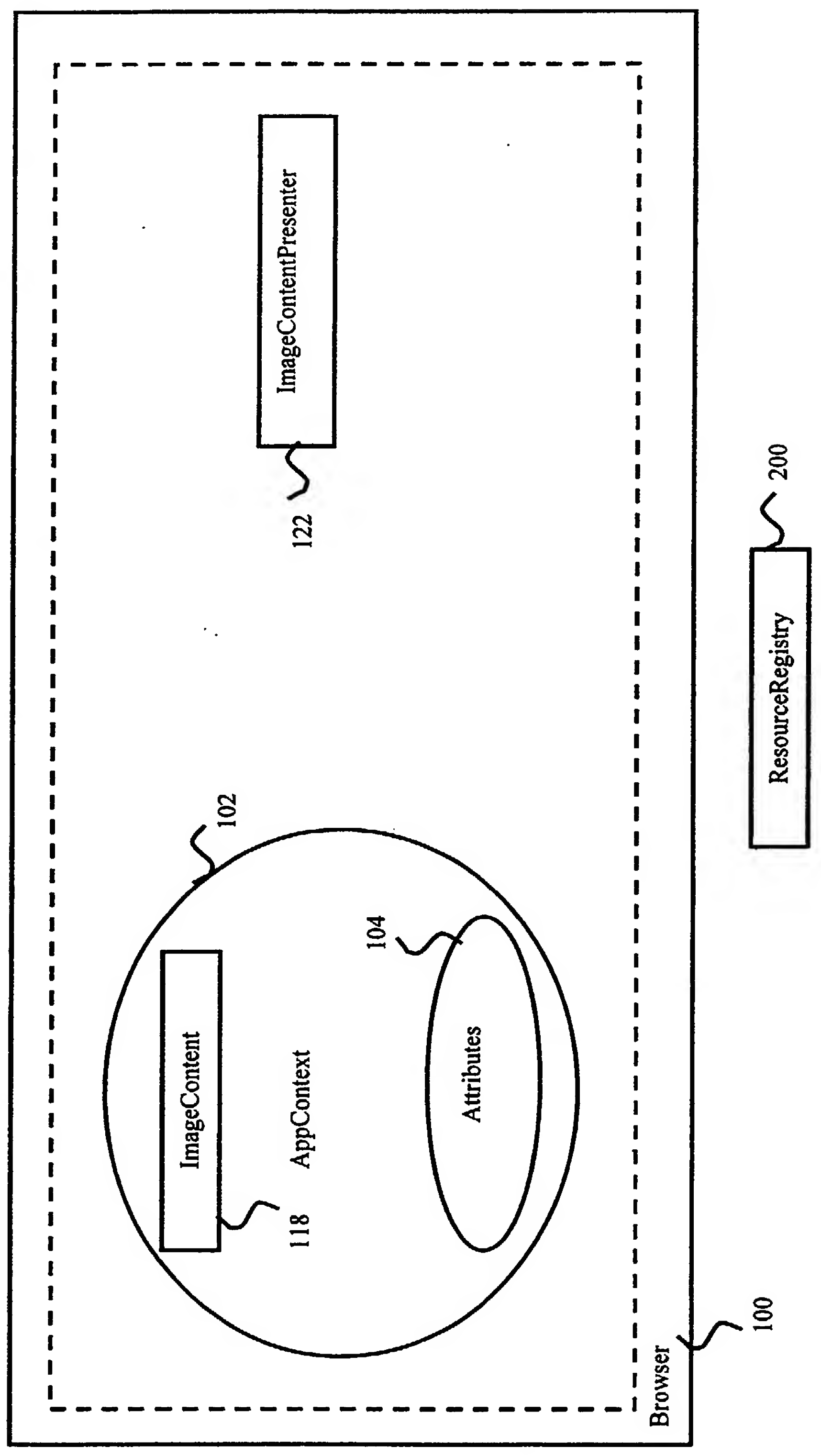

Browser
100
AppContext
102
Attributes
104
ImageContent
118
ImageContentPresenter
122
ResourceRegistry
200

Fig. 20

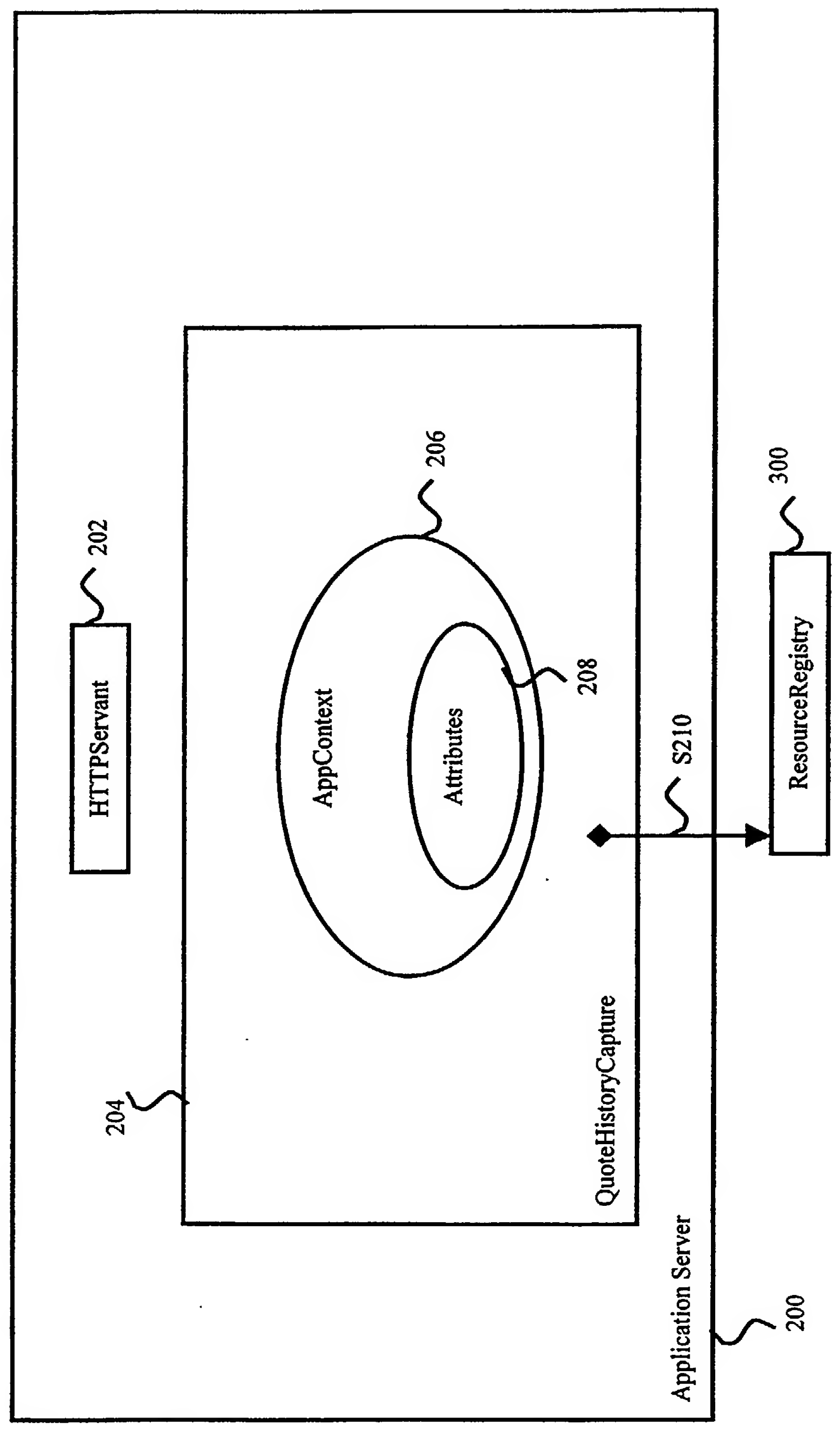

HTTPServant
202
AppContext
206
Attributes
208
S210
ResourceRegistry
300
QuoteHistoryCapture
204
Application Server
200

Fig. 21

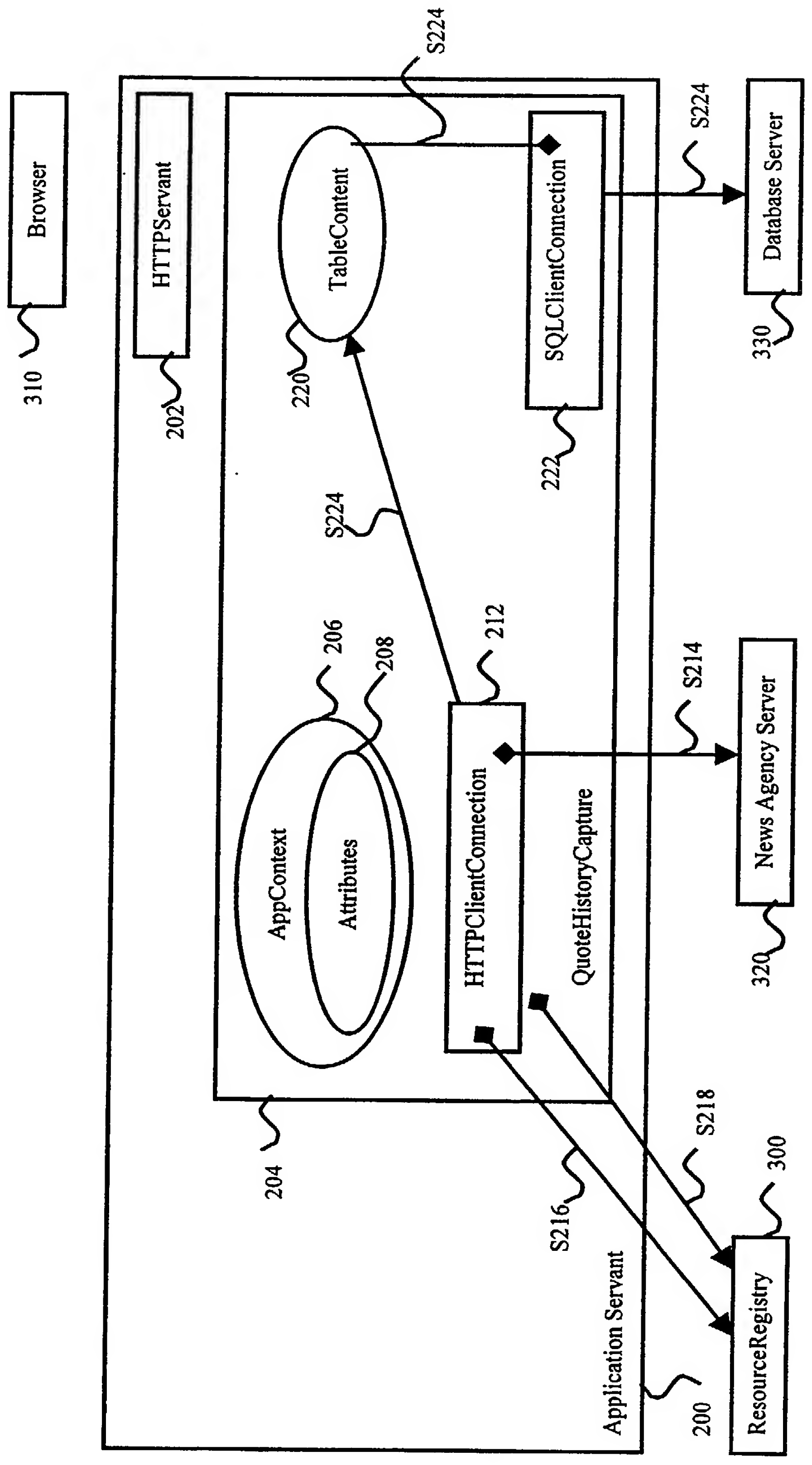

Browser
310
HTTPServant
202
TableContent
220
SQLClientConnection
222
S224
S224
S224
Database Server
330
AppContext
Attributes
206
208
HTTPClientConnection
212
QuoteHistoryCapture
S214
News Agency Server
320
204
S216
S218
300
ResourceRegistry
Application Servant
200

Fig. 22

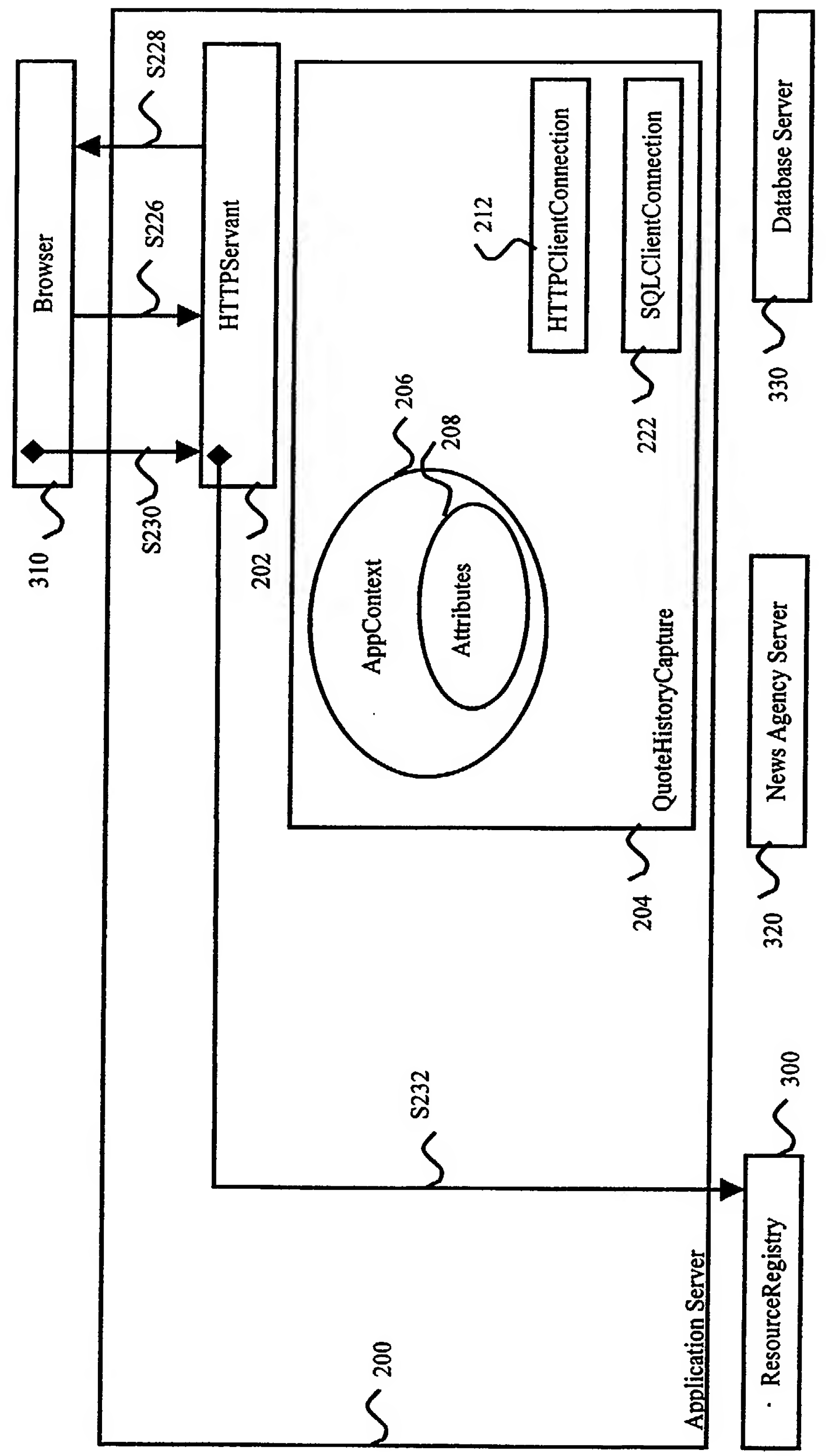

Browser
310
S228
S226
S230
HTTPServant
202
HTTPClientConnection
212
SQLClientConnection
222
AppContext
206
Attributes
208
QuoteHistoryCapture
204
Database Server
330
News Agency Server
320
S232
ResourceRegistry
300
Application Server
200

Fig. 23

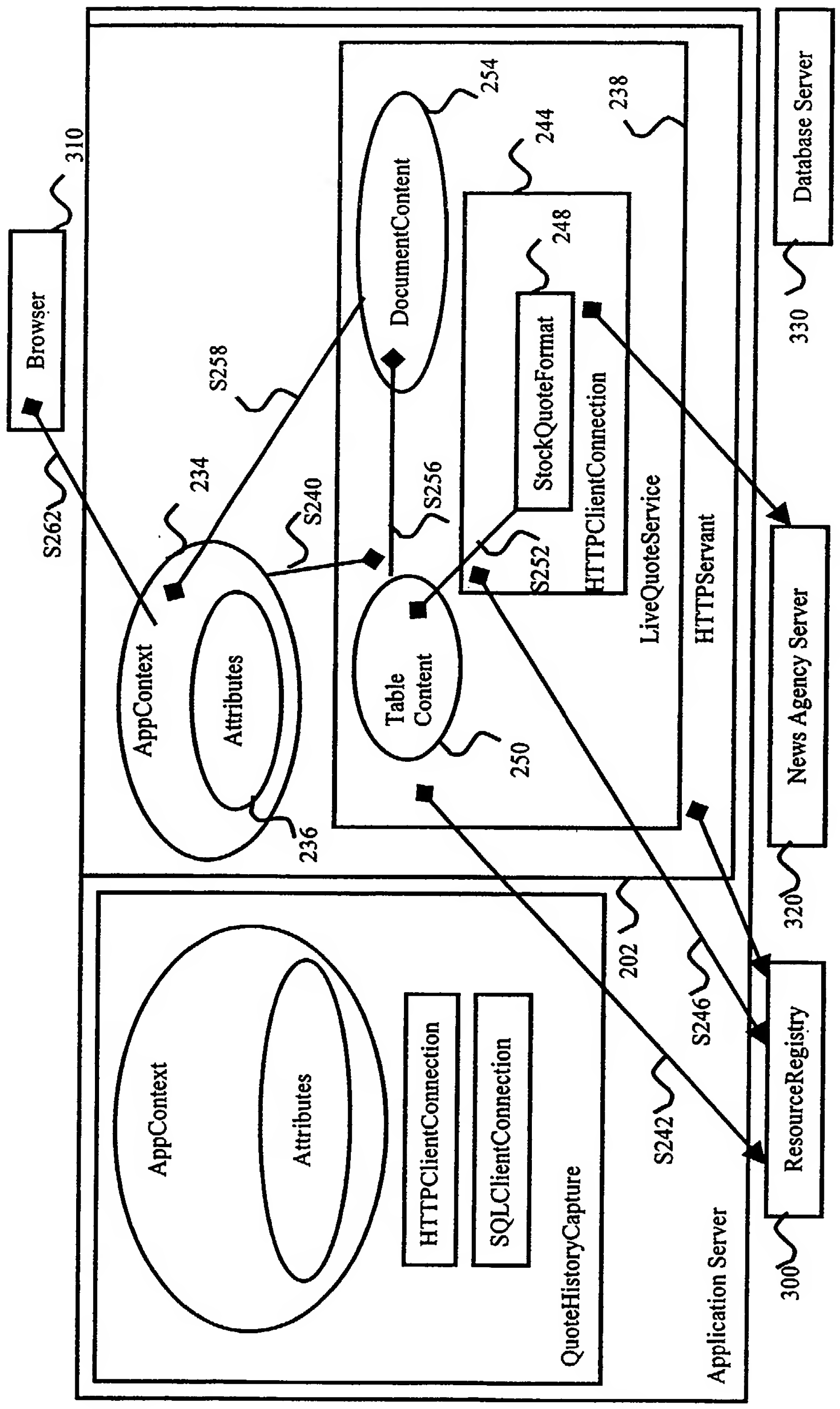

Browser
310
S262
AppContext
Attributes
234
236
S258
S240
DocumentContent
254
Table Content
250
S256
StockQuoteFormat
248
HTTPClientConnection
244
S252
LiveQuoteService
238
HTTPServant
202
AppContext
Attributes
HTTPClientConnection
SQLClientConnection
QuoteHistoryCapture
Application Server
S242
S246
Database Server
330
News Agency Server
320
ResourceRegistry
300

Fig. 24

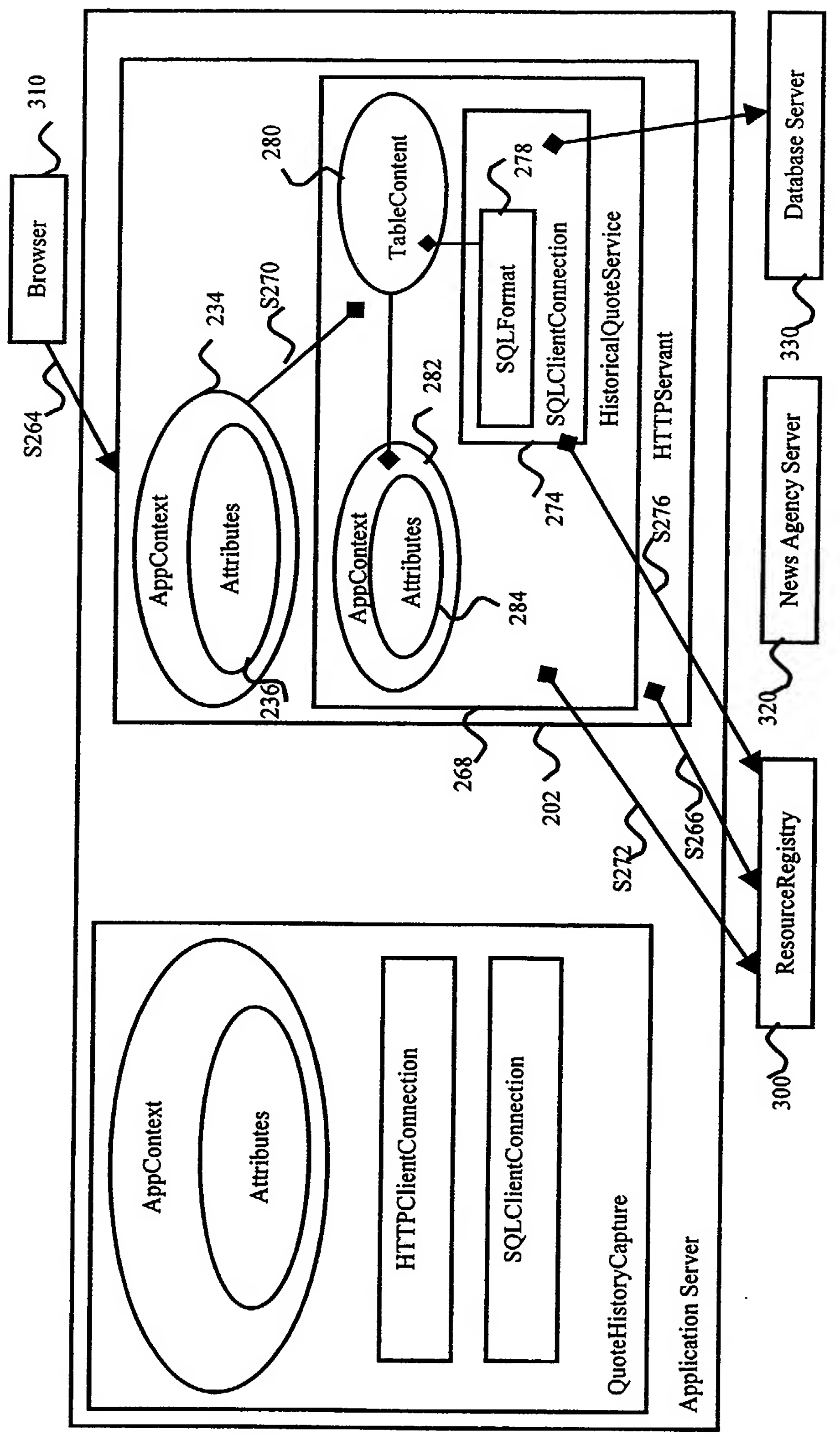

Browser
310
S264
AppContext
Attributes
234
236
S270
280
TableContent
SQLFormat
278
SQLClientConnection
HistoricalQuoteService
HTTPServant
282
AppContext
Attributes
284
274
S276
268
202
S272
S266
Database Server
330
News Agency Server
320
ResourceRegistry
300
AppContext
Attributes
HTTPClientConnection
SQLClientConnection
QuoteHistoryCapture
Application Server

Fig. 25

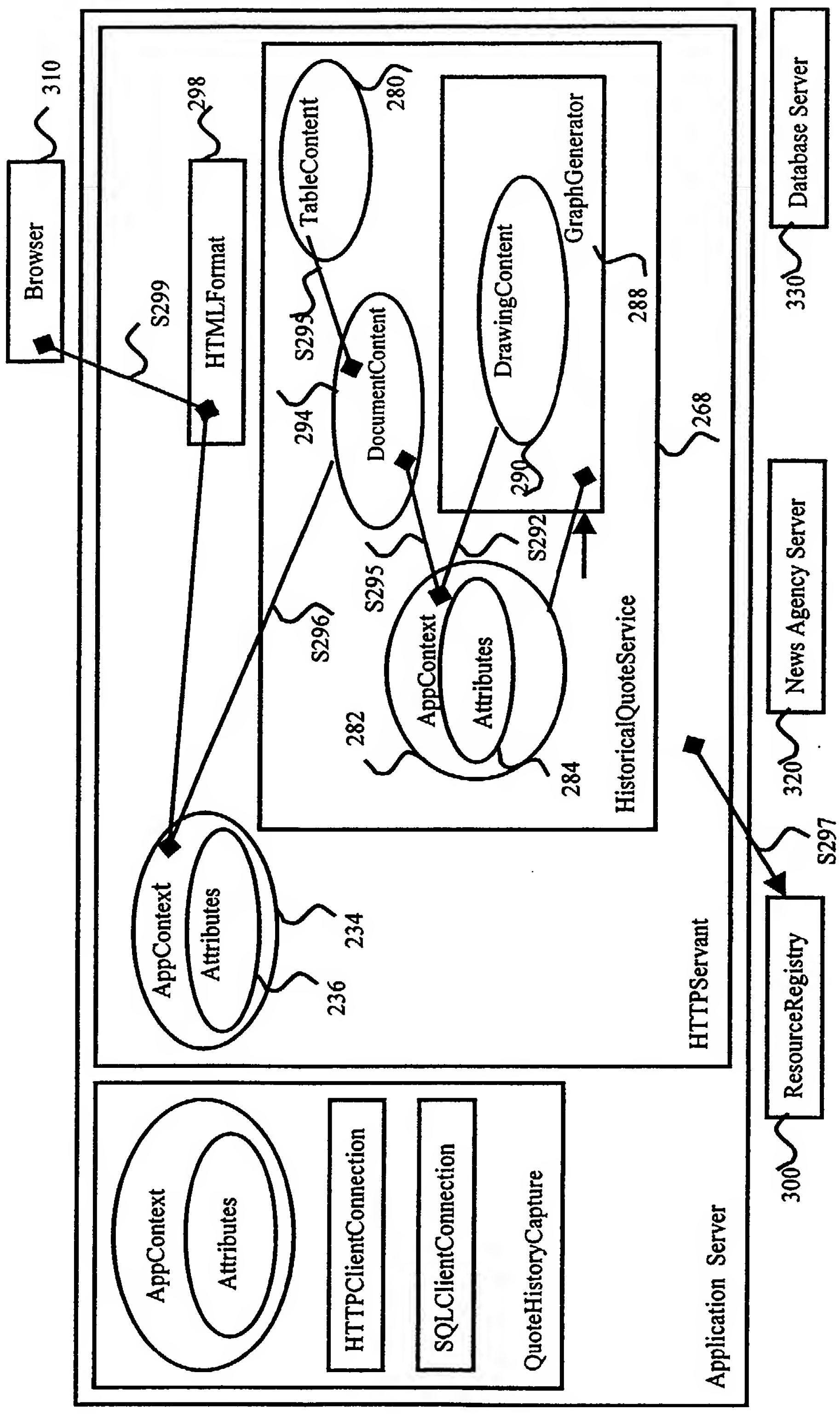

Browser
310
HTMLFormat
298
S299
TableContent
280
DocumentContent
294
S295
DrawingContent
290
GraphGenerator
288
S292
S295
AppContext
Attributes
282
284
S296
HistoricalQuoteService
268
AppContext
Attributes
234
236
HTTPServant
S297
ResourceRegistry
300
News Agency Server
320
Database Server
330
AppContext
Attributes
HTTPClientConnection
SQLClientConnection
QuoteHistoryCapture
Application Server

Fig. 26

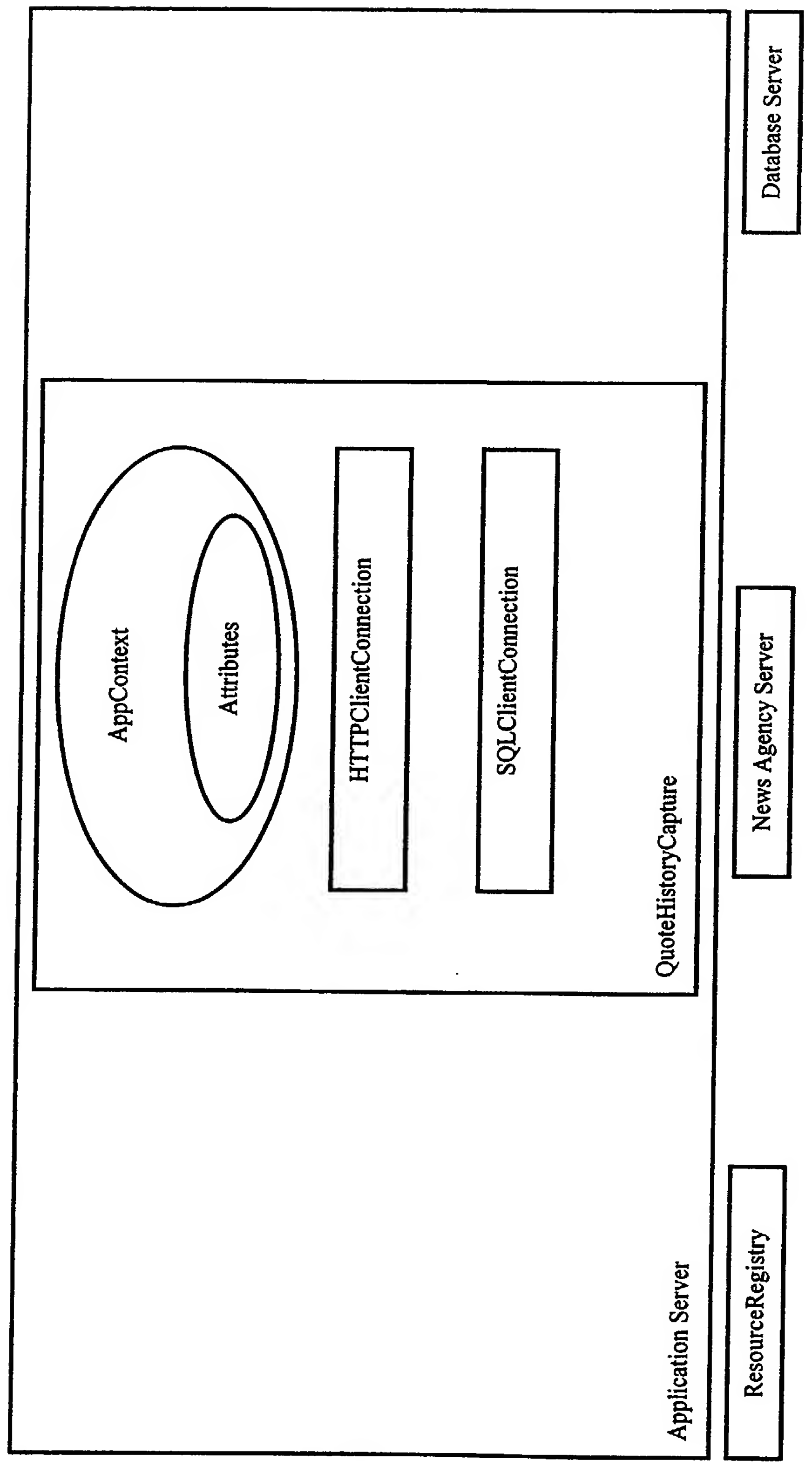

Database Server
AppContext
Attributes
HTTPClientConnection
SQLClientConnection
QuoteHistoryCapture
News Agency Server
Application Server
ResourceRegistry

Fig. 27

AN IMPROVED SOFTWARE FRAMEWORK

Background of Invention

This invention relates to a system for constructing adaptive software applications, and in particular to a software application, a method of classifying software modules, a system for synchronizing the operation of a plurality of different software modules, and a method of constructing a software application.

Computers provide means to create, manipulate and store information and transmit that information to remote locations. There are now many different ways in which data can be encoded, such as the MP3 format for sound reproduction, or the more established MPEG standard for video picture format reproduction.

Equipping a computer so that it is able to manipulate a diversity of data types and encodings is at present costly because of the proprietary and inflexible nature of the proprietary software applications currently available in the market place. Such applications are typically structured and directed to performing a particular task, such as word processing, sending e-mails, or preparing spreadsheets or charts. The function of an application is decided at design time by its developers, who then write the application code such that a variety of tools are made available to a user and a number of encodings of data are supported. The finished application is like a tool box that is equipped with tools that provide a certain functionality, but to which no more tools can be added.

Typically, a current proprietary software application supports functionalities or capabilities in addition to its principal intended purpose. For example, a word processor, in addition to its text formatting capabilities, might provide limited means for creating diagrams, or entering data into charts. However, a user is likely to be better equipped by obtaining subsequent and separate applications which are directed specifically to such extra specialized tasks, such as creating drawings or displaying data. To produce a high quality document with drawings and charts included, the user must therefore have three separate applications, create each type of data structure separately, and ultimately combine them into a finished single format. There is no guarantee that each application will be compatible with the others for importing and exporting data. Even if compatibility is not a problem, transferring data between applications can be time consuming.

Designers have addressed the problem of extending the functionality of an application by providing software interfaces which are known as Application Program Interfaces or APIs. Some applications support plug-in APIs which are software modules built specifically for that application. For example, Web browsers can often be extended to be able to display additional types of data using a plug-in; an MPEG plug-in for example would enable MPEG-encoded movies to be played within the browser. The plug-ins are however proprietary to the hosting application and so cause the application to take up even more space in memory.

Consequently, at present, the user has to carry a number of 'tool boxes' each of which is specialised to a particular task, but each of which is likely to contain tools which are identical to tools in the other tool boxes.

There is a need for a system which provides functionality to the user in an efficient way, making the best use of available resources, especially in view of the proliferation of devices for accessing the Internet, such as "intelligent" mobile phones, Personal Digital Assistants (PDAs), and digital TV set-top-boxes. Such devices, however, do not have the memory capacity or storage space to execute large, cumbersome applications.

Componentware

The concept of componentware is known and has already improved the way in which applications are developed at the design stage, and provides a partial solution to the problem of application inflexibility. With componentware it is possible for developers to combine functionality implemented as software components into tailor made applications or potentially (given some additional program code) to add functionality to an existing application via a plug-in API (Application Program Interface). The components may be developed specifically for the application or may be supplied by third party developers 'off-the-shelf'. While components are ideal for developers to extend applications for tailor-made solutions, they cannot simply be supplied to end-users to extend the functionality of their applications.

In order to build an application from components or to incorporate a component into an existing application, a developer must connect the component and the main application together using intermediate program code to act as glue. This process is known as 'wiring' components together and is often performed using an IDE (Integrated Development Environment) that allows developers to visually manipulate the components. The resulting structure is still however static and cannot be changed without modifying the 'glue' between the components.

The Java Platform

The Java Platform, developed by Sun Microsystems, provides a foundation on which the concept of componentware may be exploited. The Java Platform consists of the Java Programming Language, Java Virtual Machine, and a number of additional APIs. Java has built-in security to limit the potential for viruses and to support the level of security required in a network environment. Java is also platform independent due to its virtual machine.

The Java Programming Language is object-oriented, which means that it is able to represent real objects and imaginary concepts as software objects defined by templates known as classes. Each type of object is defined by a class, which is a template describing the state and behaviour of an object. An object's state is provided as variables or 'fields' that contain data describing the object , and an object's behaviour is provided as program subroutines or 'methods' that modify or query the state. For example, a 'Car' class might identify the state of a car by its speed and its angle of turn both expressed as numeric fields; and its behaviour would support changing gear, acceleration, and steering expressed as methods that modify those fields. Objects may work together by invoking each others methods.

Classes are templates that are used to create objects and may extend an existing class so that it 'inherits' its state and behaviour. An object is also called an instance of a class; multiple instances of a class have the same methods but may have different values in their fields. For example, a red, silk shirt, and a blue, cotton shirt may be defined by the same 'Shirt' class but will have different 'colour' and 'material' values in their fields.

Java also utilizes the concept of interfaces. An interface defines a set of methods which specify a behaviour protocol. The interface does not itself implement the methods it defines, so that essentially it is just a portable reference, and is instead implemented by classes, which thereby agree to implement all of the methods defined by the interface. Interfaces specify a protocol for communicating with an object regardless of which class actually defines the object. For example, a 'Ferrari F50' may be a subclass of 'Car' but could be described along with various other types of vehicles by a 'Vehicle' interface. An aerodynamics simulation may invoke the methods of 'Vehicle' objects, without having to be written specifically to invoke the methods of 'Car', 'Truck', and 'Motorbike' classes. Interfaces may also be extended, like classes, by other interfaces, which enables the definition of interfaces that augment the definition of any number of other interfaces.

Java has the ability to dynamically load classes and interfaces from anywhere on the network or local machine into the Java runtime environment, which allows new types of objects to be used by an application long after its development. For example, a car simulation application may have the ability to load car definitions at start-up, enabling new definitions to be included as required.

Objects in the Java environment have a so-called life cycle; they are created as a result of an action or operation, such as a calculation producing an 'answer object', and are active while they are being used. Objects that are no longer being used, are marked as garbage and periodically cleared away by the Java Virtual Machine.

The Java Virtual Machine is a specification for a computer system which may be implemented in hardware, or in software running on an existing computer system. Classes written in the Java language are compiled into instructions for the virtual machine, as opposed to a specific computer system as with other platforms. These instructions may then be loaded from anywhere, including across a network, into the Java Virtual Machine after some security checks. This permits the distribution of software across any number of networked systems.

APIs (Application Programming Interfaces) are predefined classes for common types of object used by software, such as files, network connections, and graphical user interface objects (buttons and menus for example) and may be used to extend the support and range of the Java environment.

JavaBeans

JavaBeans is an important API which sets forth a number of programming conventions known as 'design patterns' that developers may use when defining classes, so as to enable those classes to be manipulated as components by an Integrated Development Environment (IDE). A JavaBeans compliant component therefore must support all of the 'design patterns'. The patterns include a mechanism for components to notify each other that some event has occurred, the use of some fields as properties which configure or provide information about a component, and the use of accessor methods (methods prefixed with 'get' or 'set') which get and set the values of those properties.

JavaBeans includes a mechanism called Introspector which is able to provide details of how these patterns have been implemented by a specified component. An IDE can use this information to generate Java program code that connects the components or 'beans' together to form an integrated, though static, application.

In addition, Java includes the Runtime Containment and Services Protocol for JavaBeans (known as 'BeanContext'), which allows an instance of a component to be embedded or contained within another, and for the embedded component to access arbitrary services represented by some object from the containing component.

The JavaBeans Activation Framework

The JavaBeans Activation Framework ("JAF") is an API written in the Java language which supports the dynamic discovery of components that, for example, display, edit, and print a given type of data. Components are discovered based on the MIME type of the data. MIME (Multipurpose Internet Media Extensions,) is a convention for specifying the type of data, and comprises a name identifying the type of the data, such as "image" for pixel based image data, and a name that identifies the encoding format, such as "gif" for images encoded using the popular CompuServe Graphics Interchange

Format. JAF compatible components must be written for each type and encoding of data and cannot be used to handle all encoding formats of a particular data type. This limits its capability for supporting the integration of components of arbitrary functionality.

One of the Java APIs provided with the Java Platform includes support for the World Wide Web Uniform Resource Locator ("URL") standard, which is a convention for identifying resources (data and services) anywhere on the network and the user's own machine. Java allows a user to read data referenced by a URL and to send data to a location referenced by a URL. Connections made to URLs have no support for random access of data too large to fit in memory, nor support for real-time transfer of data. Such capabilities either have to be provided by the application in some way or by additional Java APIs. Representation of the data when read and decoding of its format used in transfer must also be provided by the application or by additional Java APIs. This makes it difficult for new functionality, which relies on an internal representation of data being available, to be incorporated at any time. The Java Media Framework ("JMF") goes someway to providing the ability for handling real-time delivery and other APIs being developed for Java are providing their own methods of representing and encoding/decoding data. These methods, however, are specialized in each case, for example, the Java Advanced Imaging ("JAI") API will shortly provide the ability to encode and decode images into the standard Java representation of an image. JAI will not, however, be able to handle other types of data and uses a representation of image data that is in itself limited and which requires an application to provide a lot of functionality itself, which is not suitable for dynamic applications constructed using fine-grained components.

InfoBus

InfoBus is an API for Java which enables a component to make data available and for other components to subscribe to receive that data. A component that subscribes to some data is provided with access to it by the publishing component. The publisher takes sole responsibility for managing and representing the data in a form internal to the Java Virtual Machine. When InfoBus is combined with JAF, it is possible to develop data viewing and editing components that can exchange data. This is similar to the way in which a user can import "live" data into a word processor document from another application such as a spreadsheet, as the data is modified in the spreadsheet application the changes are visible within the document.

InfoBus is useful for integration between separate applications but places unnecessary burdens on components that publish data, as they must represent the data in its internal form and provide well-known interfaces to the data for it to be accessed. The lack of functionality in the current set of interfaces limits the flexibility of manipulation by subscribers. In addition, InfoBus may be used to access multiple items of data within a group of applications rather than providing access to the specific data being used by a co-operating set of components. For these reasons, it is not possible to simply autonomously "plug" a component into an application and for it to access the appropriate data.

Related Technologies

Other non-Java technologies have been developed to support 'plug-and-play' of data and functionality, but suffer from many limitations similar to JAF and InfoBus. Examples of these technologies include OpenParts from the X/Open Group (similar to JAF), OpenDoc from Apple Computer (similar to JAF with an architecture for supporting compound documents), Object Linking/Embedding (OLE) from Microsoft (similar to JAF), and ActiveX from Microsoft (similar to JavaBeans).

The development of the World Wide Web has seen a number of new technologies being implemented. One of which is the Extensible Mark-up Language (XML) which enables different types of data to be expressed in a language that is neutral to any particular application. It is intended to overcome the problems encountered when a user attempts to load data produced by one application into another application that does not support the way in which the first application encoded the data. In addition, the Document Object Model (DOM) has and is being developed which defines a unified method of accessing and manipulating the elements of some data encoded in XML or HTML. DOM may be implemented by applications to 'expose' the data they are working on to plugged-in components, or by a document parser which handles encoding and decoding of XML/HTML files on behalf of another application. DOM is not intended for representing data not encoded in an XML manner and requires applications using it to understand the structure and elements of the encoded data as opposed to generically representing any application data.

Summary of the Invention

The invention is described in the independent claims to which reference should now be made. Advantageous features are set forth in the appendant claims.

A preferred embodiment of the invention is described below with reference to the drawings. This embodiment advantageously allows components with specific functions to be supplied to end-users, for those components to be loaded and connected and disconnected in real-time to form a dynamic application, with a continuously changing structure that reflects the needs of a user and which makes the best use of computing resources. The preferred embodiment takes the form of a software platform which supports the real time construction and modification of applications comprised from a plurality of software modules. Software modules are implemented to perform the smallest possible role or function within an application so that there is as little as possible overlap between the functionality provided by different modules. As a user of the application, or other software modules of the application request some functionality the software modules which provide that functionality are loaded in and connected to the application. Modules which provide functionality that is no longer needed are disconnected from the application so that the application takes up as little space in memory as possible.

In the preferred embodiment the platform is provided by a number of Java APIs which extend the functions provided by Sun Microsystem's Java platform. The classes and interfaces of the APIs allow software modules to be classified so that they may be integrated into an application in a known manner, and so that they may be identified easily and requested for use; they also provide a Registry of available software modules that may be used with the application, a data model for representing data in a uniform way within an application and a communication protocol for sharing data and configuration information between software modules.

Brief Description of the Drawings

The invention will now be described in more detail and with reference to the drawings, in which:

Figure 1 is a schematic illustration of the system;

Figure 2 is a hierarchy diagram illustrating the Resource Registry API of the preferred system;

Figure 3 is a flowchart which illustrates the use of the Resource Registry;

Figure 4 is a hierarchy diagram illustrating the Actor API of the preferred system;

Figure 5 is a flowchart illustrating the lifecycle of an Actor;

Figure 6 is a hierarchy diagram illustrating the AppContext API of the preferred system;

Figure 7 is a schematic diagram which illustrates possible configurations of Actors with AppContexts utilizing containment;

Figure 8 is a schematic diagram which illustrates the interactions between an Actor and an AppContext;

Figure 9 is a hierarchy diagram illustrating the Node and Content API of the preferred system;

Figure 10 is a hierarchy diagram illustrating the connection API of the preferred system;

Figure 11 is a hierarchy diagram illustrating the Format API of the preferred system;

Figure 12 is a hierarchy diagram illustrating example implementations of each of the four types of Resource provided by the APIs of the preferred system;

Figures 13 to 20 illustrate a first example use of the preferred system; and

Figures 21 to 27 illustrate a second example use of the preferred system.

Description of the Preferred Embodiment

Introduction

The preferred system comprises a set of protocols that allow a plurality of separate software objects or components, each implementing a specific function, to be combined at runtime to provide a coherent, flexible application. The software components that comprise the application are only those required to meet the present needs of a user; as a user indicates the functionality he requires, by making selections from the options presented to him by the present software objects, he causes new software objects which provide that functionality to be loaded and added to the application. Conversely, when the functionality that a software component provides is no longer required, the software component is removed from the application and disposed of. Thus, the composition of the application changes to tailor the provided functionality to that required by the user and disposes of any functionality that is not required. The application provided by the preferred system takes up considerably less space than that required by applications which are fixed at design time to provide a specific functionality and which thereafter may not be altered. It should be noted that the preferred system provides a suitable framework in which such an application can be built and a set of protocols to which software objects to be used as components of such an application must conform. It does not provide the software components themselves. These may be supplied by software develops as and when the need for a particular functionality arises.

The preferred embodiment of the invention is a set of classes and protocols define by the Java Programming Language. The runtime environment in which an application is constructed and the protocols used to support its operation are implemented by program code written in the Java Programming Language and run on the Java Virtual Machine. Use is made of various of Java's Application Program Interfaces (APIs). The software objects that provide the functionality of the application may be defined by code written in the Java Programming Language and implemented as Java objects.

The preferred system provides a number of classes of Java object to represent application data and an interface to deal with these data classes in a unified way. Single classes are provided for each binary type of data, such as text, images, audio etc. Different encodings of data within each binary type, such as GIF, JPEG, Bitmap etc for images are decoded into the single class provided for encodings of that type. Thus, as data may be represented as an instance of a common internal data type, objects that manipulate data of a given binary type need only have the ability to manipulate the internal data type provided by the system.

The preferred system also provides a number classes and interfaces in the Java Programming Language, which provide an operating framework for the Java objects that embody the application's functionality. These interfaces ensure that, wherever possible, the functions an object provide are 'fine-grained', that is are directed to performing only a single task, or a single coherent group of tasks. This ensures that the overlap between the functionalities of objects designed to perform different tasks is kept to a minimum, thereby avoiding repetition or redundancy and saving memory space. In the normal way, Java objects are loaded by the preferred system as and when they are needed, and are discarded once they are no longer needed, so that the application takes up a minimal amount of memory space.

It is intended that the Class information defining the Java objects from which application be constructed be provided by software developers supporting the preferred system. The preferred system itself, provides the means for storing such class information in the form of a registry. New Class information defining Java objects with new functions may be added to the registry by software developers or by a user to keep the registry up-to-date.

To summarise, the preferred system provides:

1) classes and interfaces that define and support the use of a plurality of Java objects that provide application functionality to a user;

2) classes and interfaces that define and support the use of a preferred data type;

3) classes and interfaces that define and support means to decode data into the preferred data type and encode data from the preferred data type;

4) classes and interfaces that define and support a registry and which allow the class information for Java objects for use with the application of the preferred system to be stored and retrieved on request;

5) classes and interfaces that define and support a mechanism for presenting the application data to the Java objects that manipulate it and for representing the attributes of an application.

Although the constituents of an application of the preferred system will all be Java objects, for the sake of clarity it is desirable to develop a nomenclature that will distinguish between the different Java objects used and which will identify their roles. For this reason, a glossary of terms used in this specification follows.

The Java objects from which the application is constructed are referred to as 'Resources'. Class information for Resources is held in a Resource Registry and may be retrieved on request when that Resource is required for use in the application. There are four different types of Resource. Those Resources called 'Actors' are those which embody or implement the functionality of the application, while those called 'Contents' represent the application data. Resources called 'Connections' and 'Formats' are used to convert data between the representation in a Content and an external data format.

The Resource Registry and each of the four different types of Resource mentioned are defined by a series of classes and interfaces organized into five Java APIs. These are the Resource Registry API, the Actor API, the Node and content API, the connection API and the Format API. Each of these is illustrated by a hierarchy diagram and will be described in detail later.

In the preferred system Resources are connected together at run-time into an adaptive and coherent application by means of containment and the use of objects called 'AppContexts'.

Containment is a mechanism provided by the JavaBeans BeanContext API and is used to embed Java objects within one another. In the preferred system this mechanism is used to contain Actors and other Resources within a 'root-container' or within containing Actors or in order to construct an applications with a coherent structure. Actors contained within a containing Actor will normally do so on request by the containing Actor in order to increase its functionality.

'AppContexts' encapsulate all or part of the application data, referred to as application state, and a set of attributes that represent the application's configuration. An 'AppContext' is managed as a service by a container and is used by Resources embedded within that container to manipulate the state and configuration of the application. The data within an AppContext is commonly represented as internal data by implementations of the Content interface.

The AppContext service is also provided by a Java API called AppContext API. This is also illustrated by a hierarchy diagram and will be described in detail in a later section.

A glossary of these terms and of others used in the description of the preferred system is included next for reference.

Glossary

Component

A component is any modular block of code which facilitates one or more functions. Components of the preferred system are implemented as objects in the Java programming language.

Resource

A Resource is a component whose class may be discovered upon request from a ResourceRegistry. A request identifies the criteria of the required Resource, which is determined by the application or another Resource when the need arises. Once a Resource class is discovered it may be instantiated and used by the application or Resource that requested it. The preferred system currently supports four types of Resources called Actor, Connection, Content, and Format, although the mechanism is extensible and allows other types of Resources to be supported as they become available.

ResourceRegistry

A component, defined by a class called 'ResourceRegistry' provided by the preferred system, that maintains information about the Resources available to an application of the preferred system, and which provides methods for searching for Resources in response to a request. The ResourceRegistry, uses a database or store into which code defining Resources may be installed. The database or the store is provided by and accessed by an implementation of an interface called ResourceRegistryProvider.

ResourceRegistryProvider

Implementations of an interface called 'ResourceRegistryProvider', as provided by the preferred system, that provide a database or store for a ResourceRegistry and methods of accessing that database or store. Different implementations of ResourceRegistryProvider may exist for different methods of storing and organizing information about Resources.

Actor

A type of Resource that provides one or more functions of an application which may be used directly by the user within a client-side application, or used as a distinct service within a server-side application. Preferably, Actors are as fine-grained as possible, that is they provide a single specific function, so that multiple Actors may be combined to provide a set of discrete functions or a single composite function. Each implementation of an Actor must implement the Actor interfaces defined by the preferred system and a single 'Role' interface. Applications are composed from Actors by means of a containment hierarchy, where an Actor provides a composite function or a set of functions implemented by Actors it contains. For example, an Actor that serves as a document editor would discover and contain Actors that manipulate a document. The Actors that manipulate the document extend the functionality of the document editor. Containing Actors discover and use other Actors in order to provide their functionality without all the functions being built into a single monolithic component.

Role

Each implementation of an Actor must implement an interface that identifies the role of the Actor in an application, such as a data 'viewer' or a 'tool' that manipulates data. Actors are discovered by their role and the criteria specific to that role. Each role is defined by a 'Role' interface of the preferred system called. Only one 'Role' interface may be implemented by an Actor.

Content

Implementations of the Content interface, defined by the preferred system, are low-level Resources that are used to represent internal data within applications, such as text, images, databases, spreadsheets, and movies.

Connection

Connections are low-level Resources that are used by Actors to transfer data between an internal data object ('Content') and another location, such as a Web server or file. Connections also manage the conversion between Content and formats external to the preferred system, such as GIF and MP3. All Connections must implement the interface named Connection, defined by the preferred system.

Formats

Implementations of the Format interface, defined by the preferred system, are low-level Resources that are used by Connections to encode internal data for transfer through a Connection and to decode data received through a Connection and encoded in a format external to the preferred system into internal data. Each implementation of Format encodes and decodes data of only a single external type.

Internalization

The process of decoding data from its external encoding format, such as JPEG, into data suitable for representation within a corresponding Content, such as ImageContent. Data to be internalized is commonly read from a Connection.

Externalization

The process of encoding data from a Content, such as DocumentContent, into a compatible external encoding format, such as HTML. Externalized data is commonly written through a Connection to some external location such as a file or a web server.

AppContext

An AppContext is an object that is issued to Actors by their container and which encapsulates data, commonly as a single Content, and a set of key-value pairs which serve to represent configuration and control information within an application of the preferred system.

Container

An object which may contain zero or more other objects. A standard interface in the Java Platform, called Collection, defines a common protocol for using containers. The JavaBeans BeanContext API extends this protocol to support the issuing of arbitrary services, represented by objects, to the objects encapsulated within a container. The preferred system provides a stand-alone software program or an extension to an existing application, called an Application Environment. The Application Environment implements the Collection interface and acts as a root-container for containing the Resources which make up an application and for providing an environment in which Actors may operate. Components an Actors of the preferred system may also implement the Collection interface to provide an environment within which Actors may operate and augment the functionality of the container.

Application

An application, as provided by the preferred system, is a dynamic containment hierarchy of Actors. The 'root' container or 'Application Environment', is a stand-alone software program or an extension to an existing application which provides some basic functionality, such as the ability to obtain and send files. The functionality of the 'root' container is extended by Actors that are discovered and activated on demand in response to the user of the application selecting operations supported by that container. Actors within the root container may then discover and activate other Actors, again in response to user actions. Applications are therefore constructed using Actors that discover and contain other Actors that provide more specialised functionality to the container. The way in which an Actor determines the need for another Actor is implementation specific, while the preferred system provides a common framework and protocol for enabling the dynamic construction of an application.

Reference will now be made to Figure 1 which shows a schematic illustration of the preferred system. The foundation for the system is a processor on which the Java platform 2 and the component DNA 4, that is the APIs described later in the specification, are installed. The system also has access to a number of Resource Registry implementations. One of these is local Registry 6, which is held in the memory of the computer processor or a local filesystem, and one is distant Registry 8, which may be held on a network. The Registries contain information about the Resources which may be used in the application. It is not necessary to have more than one Registry and the system may operate if access is restricted to just one of the two Registries mentioned.

Two application environments are also installed in memory, Applet Environment 10 and Server Environment 12. The Applet environment provides interfaces for a user to interact with in order to read data in and out and construct an application for manipulating that data. The server environment on the one hand is used to receive instructions from a web browser in order to construct a serverside application. Both environments act as containers for the Resources forming the application. Examples of use of both environments for constructing dynamic applications are found at the end of this specification. The whole routine environment of the system can be contained within a very small amount of memory, this being of the order of 50 kB. The system could equally run off the processor of a mobile phone.

In addition to the fundamental components described above is the code defining the Resources which are to make up the application. These may be provided by a developer implementing the APIs of the preferred system. Some example implementations are shown in the diagram in a number of packs which may be made available to a user of the system.

The Resource Registry and Resource Discovery

The preferred system provides a mechanism called a Resource Registry, in which software components for use in applications can be stored and from which they can be retrieved when required. The software components that are stored in the Resource Registry are referred to as Resources, and come in four flavours known as Actors, Contents, Connections and Formats. The Resource Registry provides the ability to search for Resources based on the information by which they are described. The process of searching the Resource Registry for a particular Resource or type of Resource is known as Resource Discovery.

The Resource Registry mechanism is provided by the Resource Registry API of the preferred system. This is a Java API which relies on the Java language core API, the Java IO API and the JavaBeans API produced by Sun Microsystems. A hierarchy diagram for the Resource Registry API is shown in Figure 2. Referring to this diagram it will be seen that the Resource Registry API provides two new interfaces called Resource RegistryProvider and Resource RegistryAdmin, and three new classes called Resource Registry, ResourceInfo and Resource Discovery Template.

The Resource Registry class, ResourceInfo class and ResourceDiscoveryTemplate Class all extend the object class of the Java language core API. The Resource Info class and the ResourceDiscoveryTemplate class additionally extend the SimpleBeanInfo class of the JavaBeans API, and implement the serialisable interface of the Java IO API. The methods and constants defined by these interfaces and classes are described below.

The Resource Registry is represented within the run time by a Resource Registry object produced from the Resource Registry class. From this point on we will drop the term 'object' when referring to instances of classes defined by the preferred system. For the sake of convenience, we shall refer to instances of classes merely by their class name.

The ResourceRegistry acts as a front end to the actual Registry in which Resource information is stored and which may be held in a location distant from that of the present runtime environment for an application. The actual Registry storage facility is provided by an implementation of the ResourceRegistryProvider Interface, which both maintains details of stored Resources and carries out searches on them. Different implementations of ResourceRegistryProvider may exist for different methods of storing and organizing information about Resources. For example, one implementation may use a local file as an index of locally installed Resources, whilst another may connect to a database to search and download Resources held within that database. This separation between the ResourceRegistry and a Resource RegistryProvider allows different Resource storage and retrieval policies to be used transparently by dynamic applications.

In the preferred embodiment, the information stored in the Registry to describe Resources is that which is defined in the ResourceInfo class. ResourceInfos are used to describe Resources. This class specifies a number of parameters, such as Resource name and description. Most importantly perhaps, is the description. Most importantly perhaps, is the Resource class parameter of ResourceInfo which takes a reference to the defining class of the Resource which the ResourceInfo describes.

This information may be stored in the Registry in any implementation specific manner. In the preferred embodiment however the ResourceInfo class implements 'Serialisable' of the Java IO API, so that the values of a ResourceInfo can be turned into a list so that it may be stored in the Registry in a table record or suchlike.

ResourceInfos are added to the Registry and removed from it using an implementation of the ResourceRegistryAdmin interface. This interface defines methods which take an instance of a ResourceInfo as a parameter.

A search of the Registry is performed by passing a query to the look up method of the Resource Registry. The query is represented by an instance of a class called ResourceDiscoveryTemplate. This class defines parameters that correspond to those of the ResourceInfo class, however, these parameters are to be filled in to specify the criteria of the search. In brief, a search is carried out by matching the parameter values in a Resource DiscoveryTemplate with the values of ResourceInfos held in the Registry. The results of a search are the ResourceInfos found to match the given parameters, and from the class information in the matching ResourceInfos, the desired Resources can be instantiated.

It will be appreciated therefore that ResourceInfos are essential to the operation of the Registry and consequently the preferred system. The developer of a Resource must therefore also provide a ResourceInfo to describe the Resource for it to be stored in the Registry and retrieved from it. More about ResourceInfos will be described later.

The hosting runtime of a dynamic application, i.e. an existing application that supports dynamic Resource plug-ins using the preferred system or a purpose-built stand-alone program that supports dynamic application, must configure the Resource Registry to use a particular Resource RegistryProvider. The provider may simply be implemented as part of the hosting runtime itself, such as a Web browser implementing a provider that uses a file to maintain an index of installed Resources. The hosting runtime of a mobile phone, on the other hand, may use information provided by the mobile network to access and configure the use of the mobile network operator's own provider. The Resource RegistryProvider used by the Resource Registry is set by invoking the setResource RegistryProvider() method of the Resource Registry class. For example:

```
Resource Registry.setResource
RegistryProvider(anInstanceOfAProvider);
```

The Resource Registry class defines a method that is invoked on the class itself, not on an instance of that class. This method, called 'lookup', is used to invoke a search of the Registry for any available Resources that match a given criteria. The criteria is specified as a single parameter of ResourceDiscoveryTemplate and information about the matching Resources is returned as an array of ResourceInfos. Any software within a hosting runtime that has configured the Resource Registry with a Resource RegistryProvider, can use the 'lookup' method to obtain information about each available Resource that matches the criteria specified in the

ResourceDiscoveryTemplate. Each ResourceInfo in the array returned by the 'lookup' method describes a single Resource that matches the criteria. The methods of the ResourceInfo class may then be used to retrieve the information that describes the Resource.

The ResourceInfo class is the superclass of all Resource specific 'info' classes and defines variables for holding information that is associated with all types of Resource, such as the Resource type, the name of the class that implements the Resource, version number, and a short description suitable for displaying to the user and methods to retrieve the values of those variables for any given instance. Instances of ResourceInfo hold this information as instance variables. In the preferred system, we have identified and defined four types of Resource in use in applications. These are Actors, Contents, Connections and Formats, and are defined by subclasses of ResourceInfo called ActorInfo, ContentInfo, ConnectionInfo and FormatInfo. These classes will be considered in more detail later, but for now it will be sufficient just to mention that each class defines additional parameters for describing Resources of a particular type more specifically. An instance of ActorInfo, for example, contains parameters pertaining to Resources of the Actor type, as well as the parameters specified by the ResourceInfo class. Consequently, information about an Actor is held in the Registry as an ActorInfo in serialized form.

The subclassing of ResourceInfo to describe Resources of a particular type allow developers of Resources greater freedom in deciding how they create Resources for use in the preferred system. They may create Resources that subscribe to one of the types defined by the preferred system, or may even develop new subclasses of ResourceInfo to better describe the function of their Resource. As applications develop and begin to support new functionality subclasses of ResourceInfo may be easily added. The minimum information that a developer must provide is however that specified in the ResourceInfo class.

Developers of Resources must therefore ensure that they provide this information along with the Resources themselves so that the Resource Registry can search on this information and that matching information may be passed back and used by the software entity that invoked the 'lookup' method.

The information must be provided as a subclass of the Resource specific subclass of ResourceInfo, such as ActorInfo for describing Actor implementations. This subclass must have the same name as the Resource that it describes with the suffix 'BeanInfo', such as 'FileConnectionBeanInfo' for a Resource named 'FileConnection', for example. The Resource type specific subclasses of the ResourceInfo class define additional methods for retrieving information specific to the Resource type, such as the name of a data encoding within a FormatInfo.

The ResourceDiscoveryTemplate class defines an object that encapsulates information to be used to search the registry for the desired ResourceInfo. The ResourceDiscoveryTemplate class defines variables that correspond exactly to those of the ResourceInfo class. Defining a Resource for retrieval from the Registry involves first filling in one or more of a ResourceDiscoveryTemplate's variables with values that match those of the corresponding variables of a ResourceInfo. In the simplest case a ResourceDiscoveryTemplate might only specify the ResourceName of the desired Resource as specified in its ResourceInfo. The search involves comparing this template information to the ResourceInfo object information stored in the Registry. The ResourceInfos that match are those which describe Resources that meet the criteria specified in the ResourceDiscoveryTemplate object.

Each type of Resource for which a subclass of ResourceInfo is defined must also has an associated subclass of ResourceDiscoveryTemplate. Actors, for example, are described by ActorInfo and discovered using ActorDiscoveryTemplate. For each lookup operation, an instance of either ResourceDiscoveryTemplate or one of its subclasses must be created. ResourceDiscoveryTemplate defines a constructor through which criteria are passed as parameters, and methods for use by the Resource RegistryProvider to retrieve that information to perform a search. Subclasses of ResourceDiscoveryTemplate define additional methods for obtaining the information specific to their associated Resource type.

Both the ResourceInfo and ResourceDiscoveryTemplate classes define methods, called getLookupAttributes, for retrieving the information encapsulated by their instances as a Java Map object. The Map interface is part of the Java Platform and defines the methods for accessing an object that encapsulates key-value pairs. In the Map obtained from ResourceInfo and ResourceDiscoveryTemplate, the name of each information item is used as the key and the information itself is the value. For example, the information item "ResourceName" is the full name of the class that implements a Resource. This item is encapsulated within the ResourceInfo of each Resource implementation. This item may also be specified in a ResourceDiscoveryTemplate to search for a specific Resource implementation within the Registry. The Map obtained from a ResourceInfo always contains this item with the character string "ResourceName" as the key and the name of the class as the value. If this item is also specified in the ResourceDiscoveryTemplate, then the Registry will be able to compare that value against the corresponding value in each ResourceInfo to find a match. Most other information is compared in this way.

The Map is known as 'Lookup Attributes' and is obtained from ResourceInfo and ResourceDiscoveryTemplate objects by invoking their 'getLookupAttributes' method. Subclasses override this method to add their additional, Resource type specific, information to the Map.

While most information can be compared for simple equality, some Resource type specific information may require comparison that is specific to that type. For example, data files often contain what is known as a magic number, which identifies files of a particular type. The magic number appears somewhere near the beginning of the file but not necessarily in the same place in each type of file. The FormatInfo class which describes Resources that encode and decode data of a particular type, includes an information object the magic number that identifies that type of data and an offset value which identifies where the magic number is found in files of that type. During internalization, the type of some data may be unknown and a magic number is needed to determine the type. Since the magic number may appear anywhere, a chunk of the data large enough to include any existence of a magic number is loaded and must then be compared against the magic number in each FormatInfo. Unless the offset has the value of zero, performing any comparison without reading the data chunk relative to the offset would result in a mismatch, regardless of whether the magic number actually appears somewhere in that chunk. For reasons such as this, the ResourceDiscoveryTemplate defines a method called 'filterLookupResults' which takes an array of ResourceInfo as its parameter and returns a subset of that array. This method may be used to return the magic number of an external data Formats from their associated FormatInfo.

In the preferred embodiment Resources of type Connection are used to internalize data, that is load if from an external file into an internal representation called a Content. In order to do this the connection needs to request a Format, which is the Resource responsible for actually converting data between representations, but the connection faces the problem that it cannot request the appropriate format for decoding the data until it has determined the type of data it is to decode; in order to do this it needs to know where to look in the data file for the magic number describing the type of the data, but this information is held in the appropriate FormatInfo for that data type. The connection gets around the problem by requesting all FormatInfos of type 'internalizer' to be returned by the Registry, and then uses the 'filterLookupResults' method of the FormatDiscoveryTemplate to return the magic number offset from all the FormatInfos returned from the Registry. The Connection may then use this information to search the locations of the external data file where the magic number is held. If the magic number specified by a FormatInfo is found at the file location specified by the magic number offset of the same FormatInfo then the external data type is of the data type specified by that FormatInfo and may be decoded by the appropriate Format.

Other criteria may be used with other Resources or subclasses of Resource to identify the required ResourceInfo or subclass.

Resources are registered and deregistered with a Registry by using the Resource RegistryAdmin interface. The Resource RegistryAdmin interface defines methods called 'addResource' and 'removeResource', both of which take a ResourceInfo as their sole parameter, which are used to install and uninstall Resources respectively.

The Resource RegistryAdmin interface may be implemented by the same class that implements the Resource RegistryProvider interface, or by separate classes that comprise the entire underlying registry. Access to an implementation of the Resource RegistryAdmin interface is obtained in an implementation specific manner, but the interface defines a standard protocol for adding and removing Resources. A stand-alone application that provides its own in-built registry would also include its own installation software that accesses the Registry to install and uninstall Resources via the Resource RegistryAdmin interface. Network implementations of the Registry may also allow access to the registry via the Resource RegistryAdmin for use by installation software that is separate to the application and registry implementations.

Upon invoking the 'addResource' method, the Registry uses the ResourceInfo passed as a parameter to add information to its internal index or storage facility. The way in which the information is represented, stored, indexed, and searched within the registry is implementation specific. An implementation that uses a database, for example, may obtain the information encapsulated by ResourceInfo as a Map using the ResourceInfo's 'getLookupAttributes' method and then use those key-value pairs to populate fields of a database table. The 'lookup' method of the Resource RegistryProvider interface would then be implemented to retrieve the fields of the database table that match a given ResourceDiscoveryTemplate to reconstitute the ResourceInfo.

A hosting application or any software incorporated within it may use the Resource Registry to find Resources for particular needs specified as criteria in a ResourceDiscoveryTemplate. An entity that uses the Resource Registry is known as a 'Resource Discoverer'. The criteria for discovery may be determined by the actions of the Resource Discoverer, the environment in which it is executing, the data it has been given to work with, or by specifically supporting particular types of Resource. For example, a Connection uses the Resource Registry to discover a Format capable of decoding the type of data being downloaded through that Connection.

Resource discovery using the Resource Registry will now be described in more detail and with reference to Figure 3

Resource Discovery begins when a component, Resource or the runtime environment encounters a need for functionality. This may occur when a user selects an option from a menu or GUI to request functionality, as an automated response by a Resource to an event, such as the request for an image presenter when image data is loaded in. In any case, Resource discovery only occurs when a request for a Resource is made to the Resource Registry. When this happens and the type of Resource that is to be requested is left up to the developer of Resources to encode. The advantage of the preferred system is that it provides a Registry and a Resource classification scheme that is flexible and allows developers to place a request for the type of Resource they require rather than having to specify the exact implementation of the Resource required. This means that as a Resource of a specified type are replaced or are added to by new Resources of that type they will still be found in a search for Resources of that type.

For example, EditorResources must request ToolResources that are able to modify the type of data, such as text or image, being manipulated by the Editor. Thus in the case of an ImageEditor, the requirement is for Actors with the Tool role which are able to modify image data, such as line drawing Tools, free hand Tools, painting Tools and so on. The developer of the ImageEditor must therefore encode a request for the type of Resource the image Editor requires. The request need only specify Actor, Tool and Image data in order to be sufficient to obtain in all Resources suitable for use within the ImageEditor. At no time need a specific request for a particular Tool be made, other than if it be capable of manipulating Image data, thus as Image data Tools are replaced or added or removed from the Registry, the Image data Tools available to the ImageEditor will change. Consequently, a new Tool added to the Registry will be immediately available for use within an ImageEditor or other such Resource without any problems of incompatibility.

Figure 3 shows the process of Resource Discovery in detail. At step 30, the requesting Resource, component or application logic which we shall call the Discover, instantiates a ResourceDiscoveryTemplate or one of its Resource type specific subclasses, specifying criteria which describe the required Resource.

In step 32, the Discoverer then invokes the 'look up' method of the Resource Registry passing to that method the DiscoveryTemplate as a parameter.

The reference to the current Resource Registry is provided through the AppContext service provided by the Discoverer's container, or in the case of the host application, the reference may be contained in the host applications configuration file.

Control then passes to Step 34, in which the Resource Registry's 'lookup' method invokes the 'lookup' method of the currently configured Resource RegistryProvider passing as a parameter the ResourceDiscoveryTemplate. This second 'lookup' method is implemented to search the underlying Registry such as a database or file. Within the method, the ResourceRegistryProvider invokes the ResourceDiscoveryTemplate's 'getlookupAttributes' method in step 36 in order to extract the discovery criteria from the template.

This information is then compared against the ResourceInfo information stored in the Registry in step 38 and all matching ResourceInfos returned.

All of the returned ResourceInfos are then collated into an array and may, if necessary, be passed to the template's filterLookupResults' method which returns a subset of that array which match a specified additional criteria. This method is often used by connections when looking up formats.

Usually however, no filtering is required and in step 40 the array of 'discovered' ResourceInfos are returned to the Discoverer via the ResourceRegistryProvider's and ResourceRegistry's lookup method.

Once the results have been returned to the discoverer by the ResourceRegistry, they may be used to select and use a particular Resource. Each element of the returned array is a single instance of the ResourceInfo class or one of its Resource type specific subclasses. ResourceInfo defines a method called 'getResourceClass' which returns a Java Class object that represents the Resource's defining class. This Class object may then be used to create an instance of that Resource. Often multiple Resources that match the template are required, such as an Editor looking for all compatible Tools. Other discoverers may only need a single implementation of a matching Resource and will probably just select the first ResourceInfo in the array. The policy for selecting and instantiating Resources is discoverer specific.

By way of example Java code, the process of Resource Discovery will now be illustrated. Suppose an application has some image data of type

ImageContent and needs to use a FileConnection to save that data to a file on disk in some encoding format. The application must use the Resource Registry to discover available Formats that can encode image data. The FileConnection implements the Connection interface that defines a method called 'put' which takes the data as type Content and a FormatInfo as its parameters, and then uses the Format described by the latter to encode the former. The following code illustrates how the ImageContent is saved to a file, giving the use the choice to select a Format for the file to be encoded in:

```
// create template to discover Formats for encoding
// ImageContent
FormatDiscoveryTemplate template = new
FormatDiscoveryTemplate(ImageContent.class);
```

Execution of this line of Java produces a new FormatDiscoveryTemplate, called 'template' for convenience, in which the 'class' variable is defined to be ImageContent.

```
// discover available Formats
ResourceInfo[ ] results = Resource
Registry.lookup(template);
```

The template is then passed to the current Resource Registry as a parameter of the Registry's 'lookup' method. The results of this method are returned as a ResourceInfo called 'results'. Results is an array of information for all Formats that encode ImageContent data in some external data type. The user is next requested to specify the data format into which the ImageContent is to be saved. This is done with the following line of Java.

```
// invoke another method in the application to ask the
// user which Format they would like to use

ResourceInfo selected = askUserToSelectFormat(results);
```

The user is presented with a GUI from which he makes a selection of the external data format. The Format that will be used is returned as the FormatInfo called 'selected'. The data is then encoded and saved in a file in the selected data format. This is achieved using the following lines of Java.

```
// create a connection to save the image data

FileConnection connection = new FileConnection(fileName);

// save the image data and pass the selected info as
// FormatInfo

connection.put(image, selected);
```

The first line of Java creates a new FileConnection, called 'connection, to manage the saving of the ImageContent data to a file. The name of the file is specified as a parameter of the FileConnection constructor. The 'put' method of 'connection' is then used to save the ImageContent data, called 'image', to the file with the given filename using the Format 'selected'.

The methods provided by each of the classes and interfaces of the Resource Registry API is described below.

Resource Registry Class

public static ResourceInfo[ ] lookup(ResourceDiscoveryTemplate.rdt)

Searches the internal database/registry to find information for all Resources, where the information matches the fields of the template. The information for each Resource is returned as a single element of the ResourceInfo array.

The location of the Resource RegistryProvider may also be specified as a parameter of the lookup method, thereby allowing an application to query a Resource Registry from which other Resources were obtained, or allowing a downloaded Resource to 'call home' for more Resources.

The following methods may optionally also be included in the Resource Registry:

public Resource Registry getResource RegistryProvider()

Returns a reference to the Resource Registry in use by the current application environment or runtime.

public void setResource RegistryProvider(Resource RegistryProvider)

Encapsulates a reference to the specified Resource RegistryProvider within the Resource Registry class. This reference will be returned by the getResource RegistryProvider() method.

public java.lang.String[ ] listResources(java.lang.Class ResourceDefinition)

Lists the Resources defined by the specified class/interface that may be discovered via the registry. The list is returned as a String array, each element of which is the full name of the Resource's implementation class.

public void addResource RegistryListener(Resource RegistryListener listener)

Adds the specified listener to the set of event listeners to be notified when a new Resource becomes available via the registry (probably due to it being installed) or when a Resource becomes unavailable via the registry (probably due to being uninstalled or accessibility is changed)

public void removeResource RegistryListener(Resource RegistryListener listener)

Removes the specified listener from the set of event listeners, such that it will no longer be notified of Resource availability.

Resource RegistryProvider Interface

public void ResourceInfo [ ] lookup ResourceDiscoveryTemplate

Look up Resources in the Resource Registry for the criteria specified in the given ResourceDiscoveryTemplate. The procedure is to determine the type of the Resource, search for all classes of Resource of that type, match the lookupAtributes of the ResourceDiscoveryTemplate against the lookupAtributes of the ResourceInfo objects in the search results, and finally invoke the ResourceDiscoveryTemplate's filterLookupResults () method, passing the matched results. The result of that method is the set of ResourceInfos to be returned to the caller.

It will be appreciated that in this case, the terms ResourceDiscoveryTemplate and ResourceInfo include all of their subclasses such as ActorDiscoveryTemplate and ActorInfo.

Resource RegistryAdmin Interface

public void addResource(ResourceInfo)

Adds the Resource defined by the specified ResourceInfo to the database or store of the Resource RegistryProvider.

public void removeResource(ResourceInfo)

Removes the Resource defined by the specified ResourceInfo from the database or store of the Resource RegistryProvider.

ResourceInfo Class

The following parameters are defined by this class in order to classify the Resource and provide information about the Resource to the user.

name -	a human readable name of the Resource suitable for use as a menu item or button label.
shortDescription -	a short description suitable for use as a Tool tip.
longDescription -	a longer description suitable for use in an 'About' dialog.
ManufacturerName -	the name of the Resource's developer or manufacturer.
resourceClassName -	the fully qualified name of the Resourceimplementation class.

customizerClassName -	the fully qualified name of the Resource's customizer class.
copyright -	the Resource's copyright string
version -	the Resource's version number
majorRevision -	the Resource's major revision number
minorRevision -	the Resource's minor revision number

public Map [ ] getLookupAttributes

Returns a Map of the key-value pairs of the specified ResourceInfo.

Public Map[ ] getAdditionalAttributes -

Returns a Map of key-value pairs of the ResourceInfo.

Public Class getResourceType()

Returns the type of the Resource. In the preferred embodiment the possible types are Actor, Content, Connection and Format.

public final String getName()

Returns the name of the Resource. The name should not be too long and suitable for display as a menu item.

public String getShortDescription()

Returns a short description of the Resource. The description should be as few characters long as possible so that it is suitable for display as a Tool-tip.

public final String getLongDescription()

Returns a long description of the Resource. The description may consist of as many characters as required and suitable for display within an "About" dialog.

public final String getManufacturerName()

Returns the name of the manufacturer/developer of the Resource.

public final String getResourceClassName()

Returns the full name of the Resource's defining class.

public final Class getResourceClass()

Returns the Resource's defining class. It is advised that implementations of this interface do not encapsulate the class itself, but instead use the Resource class name to load the class upon first invocation of this method. This ensures that Java's class loader does not have to spend too much time loading classes which may not be used when discovering Resources.

public final String getCustomizerClassName()

Returns the name of a customizer component's class that when instantiated enables a user or developer to visually manipulate configuration of the Resource.

public final Class getCustomizerClass()

Returns a customizer component's class. The component may be an arbitrary JavaBean component and is normally included within the BeanDescriptor object returned by the BeanInfo's getBeanDescriptor() method.

public Customizer getCustomizerInstance ()

Returns an instance of a JavaBean component customizer that enables a user or developer to visually manipulate configuration of the Resource.

public final String getCopyright()

Returns a copyright string for the Resource.

public final int getVersion()

Returns the version number of the Resource. This number, together with the major and minor revision numbers are useful in administration of software updates.

public final int getMajorRevision()

Returns the major revision number of the Resource.

public int getMinorRevision()

Returns the minor revision number of the Resource.

ResourceDiscoveryTemplate Class

public Class getResourceType ()

Returns the type of Resource for which the ResourceDiscovery Template is to search. In the preferred embodiment there are four possible Resource types, Actor, Content, Connection and Format.

public final String getResourceClassName()

Returns the full name of the defining class of the Resource for which the ResourceDiscoveryTemplate is to search.

public Map [ ] get LookupAttributes ()

This method is invoked by Resource RegistryProvider to get a Map of the attributes which describe the required Resource.

public final Map {} getAdditionalAttributes ()

This method is invoked by Resource RegistryProvider to get a Map of any additional attributes which describe the required Resource.

public ResourceInfo [ ] filterLookupResults (ResourceInfo array)

This method is invoked by Resource RegistryProvider to filter the look up results in the array of ResourceInfos passed as a parameter.

public static final ResourceInfo[ ] createReourceInfo Array (list)

Utility method for converting a list of ResourceInfos into an array. This method is used to convert the list of matching ResourceInfos returned by the lookup method into an array suitable for passing to the filterLookupResults method.

The ResourceDiscoveryTemplate class defines the same parameters as the ResourceInfo class for the purposes of searching.

Actors

An Actor is a type of Resource which provides one or more functions of an application which may be used directly by a user within a client-side application, or used as a direct service within a server-side application. An Actor implements the Actor interface provided by the Actor API of the preferred system and provides the body of code for all the methods defined in that interface as well as the code for implementing its own function.

Referring to the hierarchy diagram of the Actor API shown in Figure 4 it will be seen that the preferred system provides the Actor interface, and classes called ActorInfo, ActorDiscoveryTemplate and Operation. The Operation class extends the Object class of the Java language core API and implements the Serializable interface of the Java IO API. The Actor interface also extends the Serializable interface of the Java IO API and additionally extends the BeanContextChild interface of the Java Beans BeanContext API. The ActorInfo class extends the ResourceInfo class defined by the Resource Registry API of the preferred system and defines additional fields and methods for storing and retrieving information specific to Actor implementations. Similarly, the ActorDiscoveryTemplate class extends the ResourceDiscoveryTemplate class defined by the Resource Registry API of the preferred system.

The use of Actors to implement the user accessible functionality of an application is central to the concept of dynamic applications. An application is dynamically constructed at runtime using a hierarchy of containers. The top-most ("root") container is the hosting application itself or a container created by it to host the dynamic parts of the application. The root container, in response to a GUI interaction or request from another program, discovers, contains, and activates Actors to perform the requested operation. The contained Actors may themselves be containers and may therefore discover, contain, and activate other Actors to complete some operation, perhaps in response to a user interaction with their own GUI.

The application data is made available to the Actors which comprise the application via a service called AppContext. This will be described in more detail in the next section, but for now it is sufficient to say that an AppContext encapsulates an items of data in the form of a Content, and that any Actors instantiated to manipulate that data do so by manipulating the Content encapsulated by the AppContext. AppContexts also encapsulate key-value pairs which are used to describe configuration information such as the directory used for a temporary file, and to facilitate communication between Actors. Typically there is no more than one AppContext per container.

Organization of an application is achieved using containment and AppContexts. An AppContext may be issued to each type of data being manipulated by the application, a set of Actors required to manipulate that data instantiated and associated with each respective AppContext, and each AppContext and set of Actors contained in a parent container or other Actor. To consider a brief example, take a document Editor for editing data consisting of both text and image data.

The document Editor is perhaps first requested from the GUI of a containing environment (the roof container) and so is retrieved from the Registry and instantiated. Some document data comprising both text and image data is then loaded in using the document Editor's GUI. An AppContext is issued to the data, and a Viewer Actor instantiated to present the composite data to the user. The AppContext and the Viewer are both contained within the document Editor itself.

The document Editor may also contain an Image Editor and a Text Editor to separately handle the data in the composite data. Each of the Image Editor and Text Editor contain an AppContext for the text or the image data respectively and a Viewer to present it to the user. Additionally, each Editor is likely to contain Tool Actors to manipulate the data within the AppContext.

Applications are therefore composed principally from a variable set of Actors that provide functions and that are organised into a hierarchy of containers, to provide the functionality required when it is required. Actors implement the Actor interface for control purposes, obtain data from their container in the form of an AppContext and configure themselves automatically for the container they are embedded in. Other Resources, such as Contents, Connections and Formats are also used within an application but do not act as containers.

Containment of Actors is enabled by means of Java's BeanContext API.

The Java BeanContext API is composed of interfaces that define the methods that components must implement in order to contain or be contained within other implementing components. The API defines two main interfaces, called BeanContext and BeanContextChild. The BeanContext interface defines an object that is a container into which BeanContextChild objects may be embedded. Implementations of the BeanContext interface may be a component such as a Web browser or an image Viewer.

The BeanContextChild interface defines an object that may be embedded within a BeanContext object. In particular, BeanContextChild defines a method for informing the BeanContextChild object of the BeanContext into which it is embedded. This allows the BeanContextChild object to invoke the methods of its containing BeanContext. The BeanContext interface extends BeanContextChild so that containers may themselves be embedded, thus enabling the construction of containment hierarchies.

The BeanContext interface is extended in the Java BeanContext API by an interface called BeanContextServices. The BeanContextServices interface defines additional methods to BeanContext that enable an implementing container to provide "services" to its embedded BeanContextChild objects. "Services", in this context, are arbitrary objects that provide some mechanisms or functions for use by embedded objects. Example services include printing, inter-application data-transfer, and application specific mechanisms. A BeanContext container that implements the BeanContextServices interface shall be referred to from now on as a BeanContextServices container. Services are identified by their implementing class and are provided to BeanContextChild objects by their containing BeanContextServices, by means of an object that implements a Java interface called BeanContextServiceProvider. Instances of BeanContextServiceProvider are registered with a BeanContextServices container. Registration is achieved by passing an instance along with the class of service it provides to the container's 'addService' method, defined by the BeanContextServices interface. A BeanContextChild may request a reference to a service object of a particular class by invoking its container's 'getService' method, again defined by the BeanContextServices interface. The 'getService' method takes a class of service as its sole parameter and then checks the set of registered BeanContextServiceProvider objects for one which provides a service of that class. The container then invokes the provider's 'getService' method, defined by the BeanContextServiceProvider interface, to obtain an instance of the service, which is then returned to the requesting BeanContextChild object. If a container does not have a registered provider for the requested class of service, then it may delegate the request to its own containing BeanContextServices object, by invoking that container's 'getService' method.

Central to the dynamic construction of applications using a containment hierarchy of Actors, is the AppContext service defined by the preferred system by an interface called AppContext. This will be described in more detail in the next section. AppContexts are used to encapsulate the application State, commonly data, that is being used by a particular part of the application.

To allow Actors to be embedded within one another and to enable services, particularly AppContexts, to be accessed by Actors, the Actor interface in the preferred system extends the Java BeanContextChild interface. Actors that are themselves containers ("Container Actors") are required to implement Java's BeanContextServices interface. Container Actors must either register an implementation of the BeanContextServiceProvider interface with themselves to provide at least one AppContext instance to requesting contained Actors, or delegate the request to their own container. Examples of Container Actors are image Viewer, email client, file client, and text Editor. Examples of non-container Actors are spell checker Tool (which could be embedded within a text Editor), and file finder (which could be embedded within the email and file clients to search for messages and files respectively).

It is preferred that Actors be implemented as JavaBeans compliant components. This allows them to be additionally used like current components for wiring into static applications. JavaBeans is a set of conventions that specify how the methods are implemented so that developments Tools can aid the developer in creating the program code. JavaBeans compliant components are required to define methods for setting configuration information providing data to the component and controlling the component in an implementation specific manner.

Like any Resource, Actors are described by a by a number of parameters or information items which distinguish its function within the application. These information items are set out in the ActorInfo class which extends the ResourceInfo class, and appear again in the ActorDiscoveryTemplate class which extends the ResourceDiscoveryTemplate class so that a search may be carried out based on those information items. The ActorInfo defines three additional information items namely a role, a discriminator and a category.

The general functionality implemented by an Actor is identified by its role. A role is a "marker" interface, i.e. one which does not define any methods and which simply serves to identify classes that implement that interface. These 'role interfaces' are associated with a particular convention for the valid values of discovery criteria, that may be used to discover implementations of that role.

A discoverer of Actors, such as a hosting application or a Container Actor, can specify the role during discovery to obtain information about Actor implementations that implement that role. Additional criteria may also be specified, i.e. a discriminator and category, both of which must be character string values, but the actual possible values are specific to the role. A discriminator value is the main criteria by which Actors that implement the same role are differentiated. The value of the discriminator is role specific, for example a 'Viewer' role that identifies Actors that display data, would be differentiated by the type of data they display. The discriminator value for any 'Viewer' is therefore the name of the class that defines the type of Content displayed by the Viewer, such as "ImageContent". Therefore, if an application uses the Registry to lookup an Actor that implements the 'Viewer' role and has a discriminator that matches the data that has been downloaded, the application can automatically obtain an implementation of a suitable component and plug it into the application whilst it is still running.

Category values are used to organise implementations of particular roles, for example, multiple components that implement a 'Tool' role with a discriminator value of "ImageContent" would be used to manipulate images, and may be organised into menus and submenus by means of category values. Example categories are "freehand", "area", "shape" which would be associated with the 'Tool' role and the "ImageContent" discriminator. Categories may also be used for discovery, where they further differentiate between Actors that implement the same role and have the same discriminator values, or are simply used for organisational purposes Roles and their associated criteria conventions may be explained, added to and redefined over time to meet the requirements of modern and future applications.

Each role also has an associated set of operations which are invoked using a method that all Actors must implement. Each operation is identified by name and may have additional parameters specified as key value pairs in a Java Map. Each operation commonly has default parameter values if any or all are unspecified when invoking the operation. All implementations of the same role must implement the ability to perform the operations associated with that role to ensure compatibility of the Actor within the application. Every Actor implementation must support an operation called "default" which is often an alias of the most commonly used operation, the sole operation supported by the Actor, or perhaps a 'boot-strap' operation which displays a GUI for the user to select further operations.

Some example roles for use in client-side applications are shown in the table below, together with example discriminator and category values, and operation names.

Role Interface	Operations	Discriminated by	Categories

Client	get(retrieve data)
	put(send data)
	access(either get or put)
	default(alias of 'access')
		name of transfer or storage protocol, such as "http" or "file"	"store" for storage based systems like Web servers and databases, "messaging" for email and news services

Viewer	view(display data)
	default(alias of 'view')
		name of data class, such as ImageContent	NONE

Editor	view(display data)
	edit(display and edit data)
	default(alias of 'edit')
		name of data class, such as ImageContent	NONE

Tool	default(alias of implementation specific operation)	name of data class, such as ImageContent	data class specific such as "freehand" and "area" for ImageContent

This classification scheme with its specified operations for each defined role ensures that each Actor available from the Registry is easily identifiable so that it may be easily requested and is compatible with the expectations of the requesting container or Actor.

The methods of ActorInfo provide access to information about a particular Actor in terms of its role, discriminator, category, and supported operations. The name of the role interface may be retrieved using the ActorInfo's 'getRoleName' method, the discriminator using the 'getDiscriminator' method, and the category using the 'getCategory' method. Details of the Actor's supported operations are retrieved using the ActorInfo's 'getOperations' method. Each operation is described by an instance of the Operation class, defined in the preferred system, and used to invoke an operation on an Actor (see later). The Operation class includes the name of the operation and details of supported parameters as a Map, where each key is a parameter name and the corresponding value is the default value of the parameter.

Implementations of Actors may be discovered using a subclass of ResourceDiscoveryTemplate that is associated with the Actor Resource type, called ActorDiscoveryTemplate. To discover installed Actor implementations, an instance of ActorDiscoveryTemplate is created and the discovery criteria passed as parameters to the ActorDiscoveryTemplate's constructor. The template is then passed to the Resource Registry's 'lookup' method. The results of that method are ResourceInfo objects, or more specifically, ActorInfo objects when discovering Actors. The ActorInfo objects may be used to retrieve information about the installed Actors that match the given criteria. The class that defines a particular Actor implementation may also be obtained from its associated ActorInfo object, thereby allowing that class to be instantiated as a component object that is able to be 'plugged' into the discovering application.

The methods defined by the interfaces and classes of the Actor API are described below.

Actor Interface

public void init()

Intializes and activates the Actor. This method must be invoked, usually by a container or some deployment specific control logic, before the Actor can be used. If the Actor is already active then invoking this method does nothing.

The developer of a particular Actor, implements this method to obtain all Resources and services required by the Actor in order to operate, and set the Actor's 'active' property to 'true'. In the preferred embodiment the Actor should request at least an AppContext from its container if an AppContext is available.

If the Actor is not able to obtain all of the Resources or Services at it requests but it is still able to operate then it may still be considered 'active' and perform operations. It is preferable that, if the Resources and services become available at a later time and before the Actor is disposed of, the Actor access those Resources and Services and synchronize its internal State with them if necessary.

public void dispose()

Deactivates the Actor and disposes of any Resources and services that it obtained. This method is usually invoked by the same entity that invoked the Actor's init method, namely the Actor's container or some application control logic in the application environment.

Once deactivated the Actor may not be used to perform any operations unless its init method is first invoked. If the Actor is already inactive then this method does nothing. It is preferable that invoking the dispose method stops all activity within the Actor, such as any running threads the Actor created, and terminates any outstanding operations. Any operations which involve transactions should also stop resulting in a rollback of the transaction to the State before the operation commenced.

public boolean isActive()

Returns the boolean value true if the component has been activated, false if the component has not yet been (re)activated.

public PeformedOperation performOperation (Operation, inverse)

Performs a specified one of the Operations supported by the Actor. The operation is described by an Operation object, which identifies the name of the operation and encapsulates parameters for the operation as key-value pairs. The Operation object is passed to the performedOperation method as a parameter.

This method is typically invoked by the Actor's container or by some control logic within the application environment and provides a uniform way of instructing Actors regardless of the environment.

It is preferable if that method is implemented to validate the name of the operation as being one supported by the Actor and validate the Operation parameters for completeness and correct values.

The Operations supported by the Actor are specific to its implementation and intended function. They are left to the developer of the Actor to encode.

If the operation is successfully performed then the method requires that a PerformedOperation be created which encapsulates the details of the operation and a reference to the Actor that performed the operation. The PerformedOperation is then bound to the key "operation performed" in the Attributes object of the Actor's AppContext.

The method also takes an 'inverse' parameter which is of type 'boolean'. If this parameter is set to true then the operation is inverted, i.e. the effect of the operation is undone.

public PerformedOperation PerformDefault Operation (inverse)

Performs the Actor's default operation as specified in the Actor's ActorInfo. The effect of the default operation may be inverted or undone by passing 'true' as the value of the inverse parameter in the method.

A second performDefaultOperation method, identical in function, allows specified key-value pairs to be passed as a parameter to the method to override those normally used by the default operation.

public AppContext getAppContext ()

Get the AppContext currently in use by the Actor.

public void setAppContext/(AppContext)

Sets the AppContext for use by an Actor to that passed as a parameter. This method must be called before the Actor's init method is invoked.

public void Suspend()

Causes the Actor to temporarily halt whatever Operation it is performing.

public boolean isSuspended ()

Returns 'true' if the target Actor is currently suspended.

public void Resume()

Causes a suspended Actor to continue performing the Operation it was performing.

public void setDebugEnabled(boolean b)

If the boolean value is true then the Actor must output information detailing its activities (including event notification and receipt of events) and exception catching to a log, typically through the utility Debug class of the preferred system. This information is for use during testing and may be collected automatically for use by technical staff when deploying the technology.

public boolean isDebugEnabled()

Returns the value true if the Actor is set to output information to a log. This is the same value as is passed to the setDebugEnabled() method.

ActorInfo/Class

The ActorInfo class extends the ResourceInfo class and defines the following additional parameters to describe Resources of type 'Actor'.

roleDescriptor -	a broad description of the Actor's function or Role within an application, such as 'Editor', 'Viewer', or 'Client' for example.
discriminator -	a more specific description than RoleDescriptor of the Actor's functionality. This often has values which specify the data class that the Actor can manipulate or the name of a transfer or storage facility that the Actor supports.

category	-	a more specific description than provided by 'roleDescriptor' or discriminator to fully distinguish the function of the Actor.

The following methods are defined:

public final RoleDescriptor get RoleDescriptor()

Returns the RoleDescriptor that describes the functionality of the associated Actor.

public final String getDiscriminator()

Returns the discriminator that distinguishes the associated Actor.

public final String getCategories()

Returns the category of the associated Actor.

public Map getLookupAttributes ()

This method is invoked to obtain a Map of the attributes which define the desired Actor. This method overrides the getLookupAttributes method of the ResourceInfo class.

public Class getResourceType

Returns the type of the Resource. This method necessarily returns 'Actor' and overrides the getResourceType method in the ResourceInfo class.

public String toString ()

This method overrides the toString method in Java's object class.

ActorDiscoveryTemplate

The following fields are defined by the ActorDiscoveryTemplate class.

public String Descriptor

The full name of the role interface implemented by the required component(s). If not specified, then all Actors will be searched.

public String discriminator

A string that discriminates between implementations of the role interface, often the name of a data object class. May be specified with or without roleName.

public String category

An optional name of the category into which the required implementation(s) is/are organized.

public Map getLookup Attributes ()

This method is invoked by Resource RegistryProvider to get a Map of attributes describing the required Actor. This method overrides the getLookupAttributes method in the ResourceDiscoveryTemplate class.

public Class getResourceType ()

Returns the type of the Resource for which the ActorDiscoveryTemplate is to search. This method necessarily returns 'Actor' and overrides the getResourceType method in the ResourceDiscoveryTemplate class.

public ResourceInfo filterLookup Results (ResourceInfo array)

This method is invoked by Resource RegistryProvider *to* filter the lookup results in the array of ResourceInfos passed as a parameter.

Operation Class

public final String getName()

Returns the name of the operation.

public final String getDescription()

Returns a human-readable description of the operation. This may be displayed to users through the application or to developers through an IDE.

public final Map getParameters()

Returns the parameters of the operation as key-value pairs. The Operation objects obtained from an Actor's ActorInfo object contain the default values for each parameter.

Role Interface

This interface does not define any methods and exists to provide a common super interface for all role specific Role interfaces. In the preferred embodiment there are four Role interfaces, Client, Editor, Tool and Viewer, although it will be appreciated that more Role interfaces may be defined as the diversity of functionality supported by applications increases.

Each of the four Role interfaces supported in the preferred embodiment represent specific Actor roles within an application and are described by a corresponding RoleDescriptor class.

RoleDescriptor Class

This is the super class of all Descriptor objects that describe Actor roles. In the preferred embodiment, four subclasses of the RoleDescriptor class exist, namely ClientRoleDescriptor, EditorRoleDescriptor, ToolRoleDescriptor and ViewerRoleDescriptor.

public String getRoleInterface Name

Gets the fully qualified name of the Actor's Role Interface.

public Class getRoleInterface()

Gets the class object that represents the Role Interface.

public int getVersion()

Gets the version number of the Role Interface.

public OperationDescriptor getOperationDescriptors ()

Gets the Operations that the Role must support.

public int getDefaultOperationIndex ()

Gets an index into the array returned by get Operations() that identifies which is the default operation and which may also be referred to by the alias name 'default'.

Actor Discovery and Lifecycle

Referring to Figure 5, the lifecycle of an Actor within an application will next be described. The first step 50 is to 'Discover' an Actor that may provide the desired functionality. The process of discovery is more fully described in the previous section. Discovery occurs due to the discoverer determining or requesting an Actor or Actors that provide particular functionality. A need may be determined by the discoverer having specific support for providing composite functionality, such as an Editor discovering Tools, or in response to runtime requirements, such as a request for purchasing some product arriving at an e-commerce Web server.

In the case of an Actor that implements the 'Editor' role and that handles data of type ImageContent, the Editor needs to discover installed Actors with the 'Tool' role, that can manipulate data of type ImageContent in order to allow a user using the Editor to manipulate the ImageContent data. Tool Discovery may be performed by the code of the Editor as follows:

```
// create template to discover 'Tools' that manipulate
// images

ActorDiscoveryTemplate template = new
ActorDiscoveryTemplate("Tool","ImageContent");

// discover available 'Tools'
ResourceInfo[ ] results = Resource
Registry.lookup(template);
```

The first line of the code above uses a constructor to create an ActorDiscoveryTemplate, called 'Template', that specifies the value of 'Tool' for the RoleDescriptor and 'ImageContent' for the value of the discriminator. This template is then passed as a parameter of the lockup method of the Resource Registry, with the results of the Registry search being returned as the variable 'results' of type ResourceInfo. In fact, results contains an array of ActorInfo objects describing all available Tool Actors able to manipulate Image Context. The Editor may use the additional methods of the returned ActorInfos to retrieve information about the Tools and present a menu of the available Tools to the user. Any Tools selected by the user may be instantiated in step 52 by the Editor by calling the getResourceClass method defined by the ResourceInfo class of the ActorInfo for the selected Tool.

Any instantiated Tools are then contained in step 54 in the Editor Actor, and their init methods invoked to activate them. Once activated the Tools respond directly to user input and do not require any further intervention from the Editor, unless it is to dispose of the Tools.

The following lines of Java code illustrate this process.

```
// get the class that defines the selected 'Tool'
Class selectedToolClass =
selectedToolActorInfo.getResourceClass();

// instantiate the Tool
Tool selectedToolInstance = (Tool)
selectedToolActorInfo.newInstance();

// add the Tool to the discovering Editor
Editor.add(selectedToolInstance);

// activate the Tool
selectedToolInstance.init();

// allow the Tool to respond to user interaction, i.e. do
// whatever it is supposed to do
SelectedToolInstance.performDefaultOperation();
```

The first line of this code creates a variable of type class, called selected Tool class, which takes the class information returned by the getResourceClass method of the selected Tool's ActorInfo. The next line creates an instance of the Tool using the class information and the newInstance method of the ActorInfo. This is illustrated in step 52. The new Tool is called selected ToolInstance and is passed, in the next line of code, illustrated in step 54 to the add method of the Editor in order to contain it within the Editor. The selected Tool is then activated, in step 56 by invoking its init method and finally in the last line of code is instructed to perform its Default Operation, in step 58.

Regardless of the actual functionality of the discovered Actor, the same protocol applies for using all Actors. Thus, if a discoverer does not know what specific operations an Actor supports, or even its role, it can still use that Actor since it has implemented the methods defined by the Actor interface. These methods, are listed above and will now be described in more detail.

Once an Actor has been added to a container, it must be activated before any of its operations may be used. Activation is the process of initializing an Actor, which gives the Actor a chance to obtain any additional Resources, access to services such as an AppContext (see later), make connections, and perhaps create a GUI so that the user can interact with it. Activation is achieved by invoking the Actor's 'init' method. Once that method returns successfully without raising an exception, the Actor instance is active and ready to do whatever it is supposed to do.

While active, an Actor may be suspended by invoking its 'suspend' method. This method must suspend all activity within the Actor, i.e. freeze or pause it, which is useful for debugging, applications that drive interactive content which may be paused, and for the development of mobile components that move from one environment to another across a network. An Actor may be unfrozen by invoking its 'resume' method, at which point it must continue its original activity from the point of suspension.

While active and even if suspended, an Actor may be requested to perform one or more of its supported operations. This is achieved by invoking its 'performOperation' method and passing an instance of the Operation class to that method as a parameter. If the default operation is to be performed, then the 'performDefaultOperation' method may be invoked with no parameters instead, for convenience. If the Actor is not suspended, then the operation will be performed immediately. If the Actor is suspended, then the operation will be added to a queue and upon resuming, the Actor will perform each operation in the queue in a First-In-First-Out order.

An operation is defined by the Operation class in the preferred system. An instance of this class describes an operation that an Actor can perform and is also used to request that an operation is performed. An instance of the Operation class encapsulates the name of an operation and an optional Java Map that encapsulates parameters, if any, for the operation as key-value pairs, where the key is the parameter name and the value is the parameter value. The types of parameters are specific to a particular named operation. For example, the "view" operation ,associated with the Viewer role, has a parameter called "rendering" which may have the values "normal" to display the data in full (the default value for this parameter),and "preview" to display the data as a preview perhaps within a 'file chooser' component.

An instance of the Operation class describing a single operation supported by the associated Actor implementation may be retrieved from an ActorInfo object. Operations may be associated with a role, where an implementing Actor must support all associated operations, or implementation specific and usually invoked by the Actor itself to support role-associated operations.

An operation is performed by an Actor when its container, or some other entity within the hosting application, invokes its 'performOperation' method. That method takes an Operation object as its sole parameter and uses the operation name and operation parameter Map within that object to perform the required operation. If any parameters are not specified within the Operation's Map object, then the default values identified in the Actor's associated ActorInfo are used. The default operation may be invoked either by passing an Operation that specifies "default" as the operation name to 'performOperation' or by invoking the 'performDefaultOperation' method, defined by the Actor interface. The latter method also uses the default parameters for the operation, whereas the parameters may be specified by the application when using the former method. The "default" operation may be an alias for the most commonly used operation supported by a particular role, an alias for the only operation supported by a role, or a 'boot-strap' operation that provides the means for other operations to be invoked, such as displaying a GUI for the user to select operations through menus and buttons.

Actors are intended to be used in conjunction with an AppContext, which provides the State of the application, commonly as data, and configuration attributes. AppContexts are used to focus the operations of Actors on data, rather than on any implementation specific interactions between components. Thus dynamic applications, constructed with Actors and integrated by sharing AppContexts, are data-centric allowing them to respond to the type and content of data loaded into the application. An Actor will commonly perform an operation by using the AppContext supplied to it by its container. By 'using the AppContext' is meant that the Actor may retrieve and/or modify the data values and configuration encapsulated by the AppContext.

Once an operation has been performed by an Actor, it must notify the application, i.e. the other Actors sharing the same AppContext instance, via the AppContext's 'notifyOperationPerformed' method. This has the effect of 'firing' an event, i.e. sending an event object, to each listener registered with the AppContext instance.

AppContext is a type of service that may be obtained by an Actor from its containing instance of a BeanContextServices implementation provided that container has a BeanContextServiceProvider registered with it to provide at least one AppContext instance, or the container is part of a hierarchy in which one of the containers above the Actor does provide an AppContext.

Containers may also implement their own BeanContextServiceProvider to process requests for the AppContext service by embedded Actors, or they may use a simple implementation provided in the preferred system called DefaultAppContextProvider.

An AppContext may be requested from an embedded Actor at any time while active and not suspended, but usually occurs during initialisation, i.e. upon invocation of the Actor's 'init' method. An AppContext is requested from an Actor's container, by invoking the container's 'getService' method defined by the Java BeanContextServices interface. Containers therefore supply data for their embedded Actor's to operate on. The root container of an application commonly provides the initial AppContext into which the data of the application is encapsulated. Actors within the root container operate on that data and may provide a part of that data or entirely different (but related) data to their own 'child' Actors and so on, forming a hierarchy of components that delegate more specialised functionality down the hierarchy. Examples of hierarchies that may form in a dynamic application are described later in this document.

Actors may be able to operate without being embedded within a BeanContextServices container and consequently without an AppContext being available from such a container. Such implementations of the Actor interface are still required to be embeddable within a BeanContextServices container and able to use an AppContext, but they may also be used in the same way as existing JavaBeans compliant components, i.e. "wired" together into a static (specific solution) application. In this case they must have implementation specific methods for setting the data and configuration in place of the AppContext. Actors may therefore be used in both dynamic and static applications, and may therefore be considered as JavaBeans compliant with the ability to be used dynamically. The AppContext service will be described later in this document.

Once an instance of an Actor implementation has fulfilled its usefulness, i.e. when it is no longer in use by the application, its container deactivates it and disposes of its State, in step 58. An Actor is deactivated by invoking its 'dispose' method, which must stop any in-progress or pending operations being executed by the Actor, followed by releasing access to any services or resources it holds. In particular, any services obtained from the Actor's container, such as an AppContext, must be released by invoking the container's 'releaseService' method, defined by Java's BeanContextServices interface. When an Actor that is also a BeanContextServices container is deactivated, it must also deactivate any Actors it contains. Deactivation does not necessarily mean that the Actor instance will not be used again, simply that it is no longer being used to perform operations. Deactivation allows Resource consuming services or connections to be released by an Actor until as such time the Actor instance is used again. If the Actor is not required at all, then its instance is destroyed using Java's garbage collection, in step 60.

After an Actor has be deactivated and proving it has not been destroyed, it may be reactivated simply by invoking its 'init' method again. Reactivation may occur if a container needs to use particular Actor instances again without having to re-discover.

AppContexts

"AppContexts" are classes of objects that implement the AppContext interface defined in the preferred system. An instance of an AppContext is used to represent the State of an application and key-value pairs that represent various additional attributes of the application such as configuration information. The term State is used to mean any information that the application is structured around. This will typically be application data, encoded within a Content, but it could also be Control information for example. An AppContext is said to encapsulate some State object representing some data. This simply means that the AppContext contains a reference to the State object that it encapsulates. Replacing the State object of an AppContext with another is achieved by updating the reference to point to the new State. An AppContext is supplied by a container to every Actor embedded within that container, so that the embedded Actors may operate on the application State associated with their container. AppContexts play an important role in dynamic applications, by providing a data-centric 'hub' around which a dynamic application is constructed.

AppContexts are defined and supported by the classes and interfaces of the AppContext API illustrated in the hierarchy diagram of Figure 6. This API defines five new interfaces called AppContext State, AppContext, PerformedOperationListener, KeyValueChangeListener and Attributes and five new classes called DefaultAppContext, DefaultAppContextProvider, PerformedOperationEvent, KeyValueChangeEvent and DefaultsAttributes. The AppContextState and AppContext interfaces both extend the Serializable interface of the Java IO API. The DefaultAppContext Class implements the AppContext interface and extend the object class of the Java language core API. The DefaultAppContextProvider class implements the BeanContextServiceProvider interface of the Java Beans BeanContext API and also extends the object class of the Java language core API. The PerformedOperationEvent and KeyValueChangeEvent classes extend the EventObject class of the Java Utility API; the PerformedOperationListener and KeyValueChangeListener interfaces extend the EventListener interface of the Java Utility API. The DefaultAttributes class implements the Attributes interface which extends the Map class of the Java Utility API.

These classes and interfaces will now be described in more detail.

AppContexts may be provided as a service by a container that implements Java's BeanContextServices interface, such as a hosting application or an Actor that is also a container. A container provides this service using the service provider protocol defined by JavaBeans, i.e. by registering an instance of a BeanContextServiceProvider implemented for that service, with the container via its 'addService' method. Standard implementations of the AppContext interface and Java's BeanContextServiceProvider interface are included within the preferred system, for convenience, although containers may use their own implementations if desired. The standard implementations are called DefaultAppContext and DefaultAppContextProvider, respectively. The following example code illustrates the registration of a DefaultAppContextProvider instance (an "AppContext provider") with a container. The provider, upon its own instantiation, creates an instance of DefaultAppContext automatically and encapsulates it.

```
// create an instance of the provider
DefaultAppContextProvider provider =
new DefaultAppContextProvider();

// register the provider with the container
container.addService(AppContext.class,provider);
```

A container may provide different views of the application State to different embedded Actors. For example, a document Viewer may encapsulate a Viewer for each individual item of data within the document. This means that each embedded Viewer must be able to access the data item they are to display. The document Viewer provides a separate AppContext for each data item which is issued to the associated embedded Viewer. The DefaultAppContextProvider facilitates this by allowing a container to register an AppContext instance with the associated embedded Actor instance, ensuring that when the Actor requests the AppContext service, it will be issued with the appropriate instance. Containers, however, may use their own provider implementations to achieve the same effect. The policy for selecting which Actor is associated with which AppContext is specific to a container implementation and depends on what embedded Actors are used for within that container.

An Actor requests an AppContext from its container by means of the JavaBeans BeanContext service protocol, i.e. by using its container's 'getService' method. During activation, Actors invoke this method, identifying themselves as the requestor (specified as 'this' in Java) and specifying the AppContext class as the required service. For example:

```
AppContext theAppContext = (AppContext)
container.getService(this,AppContext.class);
```

As with all cases of requesting services from a container that implements BeanContextServices, if a BeanContextServiceProvider is not registered with the container for the AppContext service then the container delegates the request to its own container. The container invokes the 'getService' method of its own container passing the same parameters as were passed to it by the Actor. If the request cannot be satisfied by the container that is the 'root' of the hierarchy, then the Actor will not be issued with an AppContext instance. When an AppContext is unavailable, Actors may either fail to activate, continue working with their own internal State and synchronize with an AppContext if one becomes available, or "fall-back" to operating like regular JavaBeans components that require "wiring". An Actor that is operating like a regular JavaBeans component must implement methods to support providing of data, configuration information, and for communication with the main application and other components.

Figure 7 illustrates three different possible configurations of Actors and AppContexts within an root container 70. Firstly, root container 70 encapsulates AppContext 72 and a number of Actors 74a, b and c with access to that AppContext. Secondly, one of these Actors 74b may be a container itself, and as such encapsulates other Actors for delegation of its operations. The containing Actor 74b does not provide its own AppContext but instead issues the AppContext it was issued with, AppContext 72, to its contained Actors 76. Thirdly, another Actor 74c also acts as a container for a number of other Actors, but in this case, provides an additional AppContext 78 to its contained Actors 79. This may occur if the containing Actor 74c extracts some part of the data in the AppContext 72 provided by root container 70, and then supplies that extracted data to its own contained Actors 79 within another AppContext 78.

As evident in Figure 7, Actors are only aware that they are contained within some instance of a BeanContextServices container and are able to request and use an AppContext from that container. The container is responsible for controlling the Actors through the Actor interface. Dynamic integration of Actors is therefore achieved by encapsulating them within a containment hierarchy and sharing AppContexts, to reflect the structure of the application's data and/or the delegation structure of functionality.

An AppContext encapsulates the State of the application visible from the providing container and a set of attributes that support inter-Actor communication and dynamic configuration of Actors.

The State of the application is commonly the data on which Actors are to operate but it might also be other information such as Control information depending on the implementation. Actors are discovered, embedded, and activated by a container to perform operations on the State provided by that container (or from an enclosing container if the request is delegated up the containment hierarchy). Operations are associated with a particular role that an individual Actor implements and include functionality such as the creation, display, editing, manipulation, and transfer of data. AppContexts are therefore used to associate dynamically integrated Actors with data rather than application specific wiring to pass and convert data between components. The implementation of the State within an AppContext must be common to all Actors that can work with that type of State, e.g. Actors that manipulate image data must all be able to operate on the same implementation of an image data object. For this purpose, the preferred embodiment provides a data model that enables common implementations for any type of application data to be used by Actors. This data model is defined by the Content and Node interfaces described later.

The State within an AppContext is represented by an interface in the preferred system called AppContextState. Actual State objects will not normally implement this interface directly, but will implement one of its subtypes, such as Node and Content (see later). The provision of a high level non-specific State interface above that of the data model provided by the Content and Node interfaces, allows sub types of the State interface to be written by developers which expand or alter the representation of data within the application or customize it to their own needs. The AppContextState interface defines a method for disposing of the State object and is invoked by the AppContext when it is itself disposed of, see later. Subtypes of AppContextState must define methods to manipulate the specific type of State they represent. For example, image data would have methods for setting and getting pixel values.

The AppContextState object held within an AppContext may be obtained from the AppContext by invoking its 'getAppContextState' method returning a reference to the current State and put into an AppContext by invoking its 'setAppContextState' method which changes the pointer to the

State of the AppContext to that passed as a parameter of the method. When the latter method is invoked, the AppContext will notify any Actors that have registered for such notification with the AppContext, that new State has been placed into the AppContext. Actors may register, in accordance with the JavaBeans event model, with the AppContext so that they can respond to new State when it is set. A browser application, for example, might discover a data Viewer Actor that is able to display the new State and then embed and activate it so that it displays that data. Changes to the State itself are notified through events applicable to the specific type of State, see Nodes and Content later in this document. The following example code illustrates an Actor obtaining the State from an AppContext and then registering itself (referred to as 'this' in Java) with that AppContext to be notified when new State is set:

```
// get the State object
AppContextState State =
appContext.getAppContextState();

// register to be notified when new State is set
appContext.addPropertyChangeListener(this);
```

As well as encapsulating the State of the application AppContexts also store a number of Attributes which are used to facilitate configuration of Actors and communication between them. Such attributes are stored as key-value pairs where the key is the name of an attribute and the value is the attribute's valve. The attribute key-value pairs are encapsulated in an Attributes object of the Attributes interface. An Attributes object is then encapsulated within an AppContext.

The actual key-value pairs contained in an Attribute are implementation specific, that is they will be defined according to the type of Actors using the Attribute object to communicate.

Attributes may be used for example, to specify configuration information such as the local directory used for temporary file, which could have the key "temp.directory" and the value "file:/temp" to represent the local file directory named "temp". Other attributes may be used for communication between Actors that have defined relationships, such as between an Editor and an embedded Tool.

For example, consider a Tool Actor embedded within an Editor Actor, where the Tool implements a pencil effect on an image being displayed by the Editor. The Tool must be able to track the mouse coordinates within the Editor's display area, which might be supported using an attribute named "mouse.location" that has the pixel co-ordinates as its value. As the mouse is moved, this attribute would be updated by the Editor and the Tool responding to the changes being notified by a JavaBeans compliant event.

The attributes are encapsulated within an Attributes object, defined in the preferred system as an interface and implemented for general purposes as DefaultAttributes. The Attributes interface extends Java's Map interface which represents key-value pairs and adds the ability for Actors to register for notification of when an attributes value is set for the first time or changed. The attributes may therefore be used for dynamic changes in configuration and for inter-Actor communication. In order to allow inter-Actor communication, the attributes are not tied to specific Actor implementations. New key-value pairs may be added over time to include more types of 'dialogue' without affecting the implementation by individual Actors. In contrast, existing communication between components by means of wiring is fixed according to the design of the application and the methods supported by each component. Attributes are expected to be defined over time and may be generic across different applications, associated with particular types of application, specific roles, particular relationships between roles, or particular types of State. They allow communication and configuration to be specified as different types of application become dynamic using the technology described by this document.

Notification of changes to an Attributes object is supported by registering an event listener object, that implements the preferred system's KeyValueChangeListener interface, with the Attributes object. When a key-value pair changes, the Attributes object notifies all registered KeyValueChangeListeners and passes them an instance of the KeyValueChangeEvent class, defined in the preferred system to describe the change, i.e. it encapsulates the key and the change to that key's value.

The following example code illustrates obtaining the set of attributes from an AppContext, registering for notification when any attribute changes, getting the value of an attribute called "temp.directory", and setting the attribute called "mouse.location" to specify the pixel co-ordinates 15,12.

```
// get the attributes from the AppContext
Attributes attributes = appcontext.getAttributes();

// register for notification when any attribute changes
attributes.addKeyValueChangeListener(this);

// get the value of "temp.directory" String
temporaryDirectory = attributes.get("temp.directory");

// set the attribute "mouse.location"
attributes.put("mouse.location", "X=15,Y=12");
```

Once the attribute "mouse.location" is set, in the last line of code above, the Attributes object will notify all Actors registered with the Attributes that the attribute has a new value. The Actor that executed the above code would also be notified since it registered before setting the attribute.

Through the use of Attributes, AppContexts are used to associate dynamically integrated Actors with configuration information and to allow Actors to pass arbitrary information between them without application specific wiring to set configuration values and enable communication.

AppContexts are also used to notify registered Actors that an Actor using the AppContext has performed an operation. This allows Actors operating on the same AppContext to synchronize their activity and for containers to monitor the operations of their embedded Actors. When an Actors 'performOperation' or 'performDefaultOperation' method is invoked, the Actor must perform the requested operation and then create an object of type PerformedOperationEvent that describes the operation that was performed. The description is usually the same as that given in the Operation object that is used to make the request. PerformedOperationEvent is a class defined in the preferred system. Once an instance of this class is created to describe a performed operation, it is passed to the AppContext's 'firePerformedOperationEvent' method, which notifies all Actors registered with that AppContext. Actors may register as listeners for PerformedOperations using the AppContext's addPerformedOperationalListener method. The following example code illustrates how an Actor uses the AppContext to notify other Actors that it has performed an operation and then registers itself with the AppContext to be notified of operations performed by all Actors (including itself) using that AppContext:

```
// create a PerformedOperationEvent from the
// Operation object
PerformedOperationEvent event =
new PerformedOperationEvent(operation);

// notify Actors registered with the AppContext
appContext.firePerformedOperationEvent(event);

// register for notification of performed operations
appContext.addPerformedOperationListener(this);
```

The first line of this code creates a PerformedOperationEvent to represent the operation, and calls it ‘event’. ‘Event’ is then passed to an AppContext's fire PerformedOperationEvent method to notify all registered listeners. Finally, the Actor that performed the operation register itself by passing a reference to itself ‘this’ to the AppContext's addPerformedOperationListener.

Once an AppContext is no longer needed, typically when the container that created it disposed of, the AppContext must itself be disposed of. This is achieved by invoking its 'dispose' method. The 'dispose' method also disposes of the State, by invoking the 'dispose' method defined by the AppContextState interface on the State object, and the Attributes by invoking the 'clear' method defined by Java's Map interface to remove all key-value pairs. Finally the AppContext's 'dispose' method must notify all Actors, registered as PropertyChangeListeners with the AppContext, that it has been disposed of and may no longer be used. The following example code illustrates an Actor registering itself for notification of changes to the AppContext, which includes disposal, and how the AppContext is actually disposed of:

```
// register for notification
appcontext.addPropertyChangeListener(this);

// dispose of the AppContext
appcontext.dispose();
```

Figure 8 is a schematic illustration of the relations between AppContext 82, which contains a State object 84 and an Attributes object 86, and an Actor 88. Interactions between the Actor and AppContext, labelled as 88 and 90 are actions such as getting and setting the AppContext State, registering as a change listener and being notified of any changes. Interactions 92 and 94 between the AppContext State and the Actor are actions such as disposing of the AppContextState, and registering as a change listener are being notified of any such change. Similarly, interactions 96 and 98 between Attributes object and Actor represent such actions as getting and setting key-value pairs, remove key-value, registering as a KeyValue Change Listener or a PerformedOperation Listener and being notified of any such change, or any such performed operation.

AppContextState interface

This interface defines the State of an application that is encapsulated within AppContext objects. Implementations are typically data objects although other implementations may exist.

public void dispose()

Disposes of the State encapsulated by the AppContext. Once invoked, the State object may not be used.

public boolean isDisposed()

Returns the boolean value 'true' if the State is disposed of, otherwise 'false'.

public void addPropertyChangeListener (PropertyChangeListener)

Registers an object to be notified when the AppContext State changes. The object to be added as a listener is described by the PropertyChangeListener object passed defined in the Java Beans API, which is passed as a parameter to the method.

public void removerPropertyChangeListener (PropertyChange Listener)

Unregisters the object specified by the PropertyChangeListener parameter so that it is not notified when the AppContext State changes.

AppContext Interface

public AppContextState getAppContextState()

Returns the AppContextState object encapsulated within the AppContext.

public void setAppContextState(AppContextState)

Encapsulates the specified AppContextState object, replacing (and disposing of) the currently encapsulated AppContextState if it exists. A PropertyChangeEvent is also passed to each PropertyChangeListener registered with the AppContext. The event has the property name "appContextState".

public boolean hasAppContextState()

Returns the boolean value 'true' if an AppContextState currently encapsulated by the AppContext.

public Attributes getAttributes()

Returns an Attributes object that encapsulates the key-value pairs that represent the shared application variables.

public void addPropertyChangeListener(PropertyChangeListener)

Registers the specified PropertyChangeListener to be notified when a new AppContextState object is encapsulated using the setAppContextState() method. PropertyChangeListener is included in the JavaBeans API.

public void removePropertyChangeListener(PropertyChangeListener

Deregisters the specified PropertyChangeListener so that it will no longer be notified when a new AppContextState object is encapsulated by the AppContext.

public void setPeformedOperation (PerformedOperation)

Sets the 'performedoperation' which identifies the last Operation to be performed on the State encapsulated by the AppContext, and issues a PropertyChangeEvent to all registered PropertyChangeListeners.

public void PerformedOperation get Performed Operation ()

Gets the performed operation property of the AppContext State.

Default AppContextProvider Class

This class defines the following fields.

Default AppContext - the default AppContext which is issued to Actors without binding.

ActorToAppContextBindings - this is a Map of Actor keys to AppContext value bindings.

The following methods are also defined.

public Object getService BeanContextServices, requestor, serviceCls, ServiceSelector

This method is specified in the Java Beans. Beancontext. BeanContextServiceProvider interface.

public Iterator get CurrentServiceSelectors of BeanContextServices, Service Class

this method is a no-op implementation, that is one that simply returns null, since in the preferred embodiment service selectors are not used with AppContexts.

public void releaseService () BeanContextServices, requestor, service)

This method is a no-op implementation since in the preferred embodiment there are no special requirements for when an AppContext is released.

public void addRequestorBinding (Actor, AppContext)

This method adds a binding to the specified Actor to the specified AppContext. This does not issue the AppContext to the Actor, but associates them so that when the Actor requests a service of typeAppContext it will receive the one specified in the binding.

public void removeRequestorBinding (ActorAppContext)

Removes a binding between the specified Actor and the specified AppContext. Any container using this AppContextProvider necessarily must ensure that after a child Actor has been removed the corresponding binding for that child is also removed.

PerformedOperationEvent Class

public final String getActorClassName()

Returns the full name of the class of Actor that performed the operation.

public final Class getActorClass()

Returns the class of the Actor that performed the operation.

public final Actor getActor()

Returns the actual Actor that performed the operation, providing it is still active an encapsulated by the same container that it performed the operation within.

KeyValueChangeEvent Class

The possible changes that may occur to an Attributes object are the putting of a key-value pair into the Attributes (key-value mapping) or the removal of a key-value pair from the Attributes (key-value unmapping).

This class defines two integer constants that are used to identify the actual event that occurred to one or more key-value pairs. These constants are defined as follows:

public static int KEY_VALUE_MAPPED

Identifies an event that occurs when one or more key-value pairs are put into an Attributes object.

public static int KEY_VALUE_UNMAPPED

Identifies an event that occurs when one or more key-value pairs are removed from an Attributes object.

The methods are as follows:

public final String getAttributeKeys()

Returns the keys affected by the change as an array of Strings.

public final Object getOldAttributeValues()

Returns the old value associated with each affected key as an array of Objects. Each value in the array is in the same position as its corresponding key in the array returned by getAttributeKeys().

public final Object getNewAttributeValues()

Returns the new value associated with each affected key as an array of Objects. Each value in the array is in the same position as its corresponding key in the array returned by getAttributeKeys(). The return values are null with events of type KEY_VALUE_UNMAPPED.

public final Object getOldAttributeValueFor(String key)

Returns the old value of the specified key if its value has changed, or throws an IllegalArgumentException (defined in Java's standard API) if the value associated with the specified key has not changed.

public final Object getNewAttributeValueFor(String key)

Returns the new value of the specified key if its value has changed, or throws an IllegalArgumentException if the value associated with the specified key has not changed.

public boolean isVetoed()

Returns 'true' if the change has been vetoed and the event has been marked as such to notify KeyValueMappingListeners of the veto.

public String idString()

Returns the id of the event as a string. The possible strings that may be returned by this method are "KEY_VALUE_MAPPED" and "KEY_VALUE_UNMAPPED".

PerformedOperationListener Interface

public void OperationPerformed (performed OperationEvent)

Notifies the listener that an operation defined by the performedOperationEvent passes as a parameter to the method has been performed.

KeyValueChangeListener Interface

The methods defined by this interface are invoked by the Attributes object when a change occurs to that object. A method is defined for each event id constant defined by KeyValueMappingEvent namely "KEY_VALUE_MAPPED" and KEY_VALUE_UNMAPPED. When an Attributes object changes, a KeyValueMappingEvent instance is created and given an event id matching one of these constants. The method associated with that event id is invoked on each registered listener with the instance of KeyValueMappingEvent as the parameter. The following methods are defined:

public void keyValueMapped(KeyValueMappingEvent evt)

Invoked by an Attributes object with which the listener is registered when a key-value pair is put into that object, or an existing key is given a new value. The KeyValueMappingEvent passed to this method must have the id "KEY_VALUE_MAPPED".

public void keyValueUnmapped(KeyValueMappingEvent evt)

Invoked by an Attributes object with which the listener is registered when a key-value pair is removed from that object. The KeyValueMappingEvent passed to this method must have the id "KEY_VALUE_UNMAPPED".

Attributes Interface

public Object get(key, defaultValue)

Retrieves the value associated with the specified key. This method augments the get() method of Java's Map interface by requiring that the specified defaultValue is returned if the key is not encapsulated by the Attributes object or cannot be retrieved through its "resolve parent".

public Iterator keyIterator()

Returns an Iterator object that permits the sequential retrieval of the Attributes' keys. Iterator is an interface defined in Java's utility API. Modification through the Iterator is also subject to the event notification and modifiability of the Attributes.

public String keysToArray()

Returns the Attributes' keys as an array of Java Strings.

public Iterator valueIterator()

Returns an Iterator object that permits the sequential retrieval of the Attributes' values.

public Object valuesToArray()

Returns the Attributes' values as an array.

public boolean contains(key, value)

Returns the boolean value 'true' if the Attributes contains the specified key-value pair.

public boolean containsAll(Map)

Returns the boolean 'value' true if the Attributes contains all of the keys in the specified Map.

A second contains All method is also defined in which a boolean Checkvalues parameter may also be passed as a parameter of the method. If Checkvalues is true then this method will return true if all of the keys in the specified Map and their corresponding values match the keys and values encapsulated within the Attributes.

public Object remove(key, value)

Removes the specified key-value pair and returns the value for convenience when processing key-value pairs as they are removed.

public Map removeAll(Map)

Removes all key-value pairs where each key is also contained in the specified Map.

A second removeAll method is also defined which takes a boolean valueMatch parameter. If valueMatch is true then each key-value pair will only be removed if the key is also contained in the specified Map along with the same value as within the Attributes.

public Map retainAll(Map map)

Removes all key-value pairs where the key is not contained in the specified Map, such that the Attributes only contain the keys in the specified Map.

A second retainAll method is also defined which additionally takes a boolean valueMatch parameter. If valueMatch is true then a key will also be removed if it has a value that differs from the same key in the specified Map.

public Map submap(String subkey)

Returns a subset of the Attributes' key-value pairs as a Map. The key-value pairs are determined by specifying a subkey (the first few characters of a key). For example, the keys "lizzard.komodo", "lizzard.iguana", and "lizzard.gecko" can all be found by specifying "lizzard" as the subkey.

public Attributes getResolveParent()

Returns the "resolve parent" of the Attributes object on which this method is invoked.

public void setResolveParent(Attributes resolveParent)

Sets the "resolve parent" of the Attributes object on which this method is invoked. In this case the 'Resolve parent' is the

Attributes object from which to obtain a value if a given key is not contained in the target Attribute objects during a call.

public boolean isModifiable()

Returns the boolean value true if the Attributes object is modifiable, otherwise returns true.

public void addKeyValueMappingListener(KeyValueMappingListener kvml)

Registers the specified KeyValueMappingListener with the Attributes object so that it may be notified when a key-value pair changes, i.e. put into or removed from the Attributes.

public void removeKeyValueMappingListener (KeyValueMappingListener kvml)

Deregisters the specified KeyValueMappingListener so that it will no longer be notified of changes to key-value pairs in the Attributes object.

DefaultAttributes Class

The methods are essentially the same as for Attributes.

protected boolean is KeyValid (Key)

Returns true if the specified key is a valid key for the default Attributes. This method is used to check the validity of a key before it is placed in the Attribute object.

protected boolean is ValueValid (value)

This method performs a check on each value as it is stored in the Attribute. By default this method simply ensures that the value is Serializable.

public boolean contains Key(key)

Returns true if the default Attributes object containing the specified key.

Public object get (Key)

Returns as an object the value associated with the specified key.

Public Object put (key, Value)

Puts the specified key-value pair into the default Attribute.

Public void putAll(Map)

Puts all of the key-value pairs contained within the Map into the Attributes.

public object remove(key)

Removes the attribute with the specified key from the Attribute.

public void clear

Removes all attributes from the Attribute.

public boolean isEmpty()

Returns 'true' if the Attribute is empty.

The rest of the methods are identical to those of the Attributes interface.

Nodes and Content

The preferred system also defines a data model, a common representation of all data types that removes the need for Actors to be able to manipulate data in the many different encodings that exist today. Instead, Connections and Formats, which will be described in the following sections, convert data from an 'External' encoding, that is one other than that defined by the preferred system, into the common 'internal' data type of the preferred system. The internal data type is represented by objects called Nodes and Contents. In particular Contents are Resources and may be discovered from the Registry. Installed Contents in the Registry represent the type of data, this being text, image, audio data and so on, that the application can hold internally.

Nodes and Contents are defined by the Node and Content API, which is illustrated in the hierarchy diagram of Figure 9. The API defines a Node interface, which extends the AppContextState interface of the AppContext API of the preferred embodiment, and a Content interface which extends the Node interface.

The API also defines three new classes called 'ContentDiscoveryTemplate', which extends ResourceDiscoveryTemplate, 'ContentInfo' which extends ResourceInfo and 'NodeEvent' which extends the EventObject class of the Java Utility API. A NodeListener interface which extends the EventListener class of the Java Utility API is also defined.

A "Node" is a type of AppContextState that encapsulates an element of data and/or child nodes that form a composite data element. 'Child' nodes are contained within a 'parent' node using the same underlying collections capability of Java that is used by BeanContexts. Nodes are defined in the preferred system by an interface called 'Node' that extends the AppContextState interface. A Node may therefore be encapsulated within an AppContext and used as a State object. Different implementations of the Node interface may exist to encapsulate different types of fine-grained data, such as a paragraph of text or a single cell of a spreadsheet.

Most Nodes are not complete data structures but are intended to be encapsulated within an object that implements the Content interface Implementations of the Content interface represent complete data structures such as a text file or an entire spreadsheet. Contents encapsulate one or more Nodes, which may form a hierarchical structure if any of those Nodes may contain child Nodes. Content itself extends the Node interface and may therefore be embedded within other Nodes. The types of Node that may be embedded within one another is governed by the containing Node, for example, paragraph nodes may only contain text nodes and spreadsheet content may only contain cell nodes.

Example implementations of the Node interface are shown in the table below:

Node Name	Represents	Type of Value	Allowed Child Nodes
TextParagraphNode	A paragraph of text	None	CharacterStringNode
CharacterStringNode	A block of text in the same style	CharacterString	
CellNode	A single cell of a spreadsheet	None	NumericNode, FormulaNode, Content
NumericNode	A numeric value	Numeric Value	
FormulaNode	An algebraic formula or expression	Character String	
PixmapNode	A single raster image frame	2D Array of pixel values	
MessagePartNode	A part of a message such as an attachment	None	Content

Example implementations of the Content interface are shown in the table below:

Content Name	Represents	Allowed Child Nodes
TextContent	Plain and styled text	TextParagraphNode
TableContent	Databases, spreadsheets, and other table structures	CellNode
ImageContent	An image document consisting of one or more raster image frames.	PixmapNode
DocumentContent	Compound documents such as Web pages and word processor documents	Content
MessageContent	Email and discussion group messages.	MessagePartNode

A vast number of implementations of the Node and Content interfaces are possible for an equally vast number of possible data types. These implementations must be common within to all runtime environments within which they are used so that Actors are able to use the same consistent representation of some data type within all applications. The Node and Content Interfaces do not define the necessary methods to support the manipulation of all the possible data types put provide a structure under which the data type specific sub interfaces may be integrated into the data model. The sub interfaces may be created by developers over time to reflect the changing types of data in use.

Query and manipulation of data is supported through methods defined by the Node and Content interfaces and by their implementation classes such as those identified in the tables above. It will be appreciated that more implementations may be added as data requirements change.

The methods defined by the Node interface are used to set and retrieve the value of a node, query and manipulate the children of a node, to set and retrieve the name of a node (used to identify a specific Node instance within a node hierarchy), and to dispose of a Node instance (see below). Methods are also defined by Node for registering and deregistering event listener objects (such as Actors) with an instance of a Node implementation, so that those listeners will be notified when the Node's value or children are changed, or when the Node has been disposed of. Listeners implement the NodeListener interface and the event object used to describe changes to the Node must be instances of the NodeEvent class, both of which are defined in the preferred system. The implementation specific methods of a Node should provide methods similar to the Node interface but specific to the value and children supported by the implementation. Additional methods for manipulating a node in an implementation specific manner, such as setting the style of text within a CharacterStringNode.

Since the Content interface extends the Node interface, all implementations of Content necessarily support the methods defined by the Node interface. The Content interface essentially serves to identify implementations as being Nodes that can be transferred through Connections and encoded by Formats (see later in this document).

In addition, the Content interface defines two methods for getting and setting the 'current external format' of a Content instance, called 'getCurrentExternalFormat' and 'setCurrentExternalFormat' respectively. The 'current external format' is the last encoding format in which the Content object was sent or from which it was decoded. The format must be specified as a MIME type string that may be used to discover a suitable Format to encode or decode that type of data. These methods allow an instance of Content to ‘remember’ what external encoding it was decoded from.

Implementations of the Content interface must define methods for manipulating instances of those implementations, appropriate for the type of data they represent. For example, ImageContent may define methods for applying an effect to multiple image frames encapsulated within an instance. This would be more convenient than using the Node interface methods to retrieve each encapsulated PixmapNode and modify the pixel values encapsulated within that node.

Since the Content interface extends the Node interface which in turn extends the AppContextState interface, instances of all Content and Node implementations may be encapsulated within an AppContext as AppContextState to encapsulate and structure the data of a dynamic application. Actors use the methods defined by the Node and Content interfaces, and the implementation specific methods to access and manipulate the data. As mentioned previously, when an AppContext is disposed of, it also disposes of the encapsulated AppContextState. When an instance of a Node or Content implementation is disposed of, it also disposes of any child Nodes and Content, which in turn dispose of their children. Any other system resources used by those objects are also released.

Implementations of Content may be registered with a Resource Registry so that available implementations can be discovered dynamically by an application. As with all Resources discoverable using the Resource Registry, Content implementations may be discovered using an instance of a class called ContentDiscoveryTemplate, defined in the preferred system as a subclass of the ResourceDiscoveryTemplate class so that it may be used by the Resource Registry for discovery. The description of Content implementations are implemented as subclasses of ContentInfo, a class defined in the preferred system as a subclass of ResourceInfo, that may be registered for discovery by the Resource Registry.

Content Interface

public java.lang.String ***getCurrentExternalFormat()***

Gets the MIME type identifier of the external format in which the target was transported and internalized from, and/or in which it will be externalized and transported as by default.

public void ***setCurrentExternalFormat*** *(java.lang.String mimeTypeAsString)*

Sets the MIME type identifier of the external format in which the target was transported and internalized from, and/private in which it will be externalized and transported as by default.

This method is used by InternalizerFormats after creating and internalizing the target, to identify the format of the external representation.

Public boolean is ExternalState()

Tests whether the content is actually external and being internalized on demand, i.e. by using an ExternalNodeHolder.

Public boolean isModifiable()

Tests whether the content can be modified. Modification may be prevented if the state is external and cannot be updated. Note that all Content created internally, i.e. not as a result of internalization is modifiable by default.

Node Interface

*public void **clear()***

Clear the value and children of the target

*public java.lang.String **getName()***

Get the name of the target, if available. It is not required that a Node instance has a name.

*Public void **setName** (java.lang.String name)*

Set the name of the target. A Node implementation or specific instance may impose a naming convention which must be adhered to for this method to complete, otherwise an IllegalArgumentException is thrown.

*Public java.lang.Object **getValue()***

Get the value of the target.

*Public void **setValue** (java.lang.Object value)*

Set the value of the target. The value musts be valid for the target, otherwise an IllegalArgumentException is thrown.

Public java.lang.String ***getValueAsString()***

Gets a string representation of the target's value. This may be equivalent to getValue().toString() but should not be assumed to be the case.

Public java.lang.String ***getValueAsPresentableString()***

Gets a human presentable string representation of the target's value, i.e. a character string that may be displayed to a human user. The string is not guaranteed to be different to that returned by <@linkgetValueAsString()>.

Public <u>Node</u> ***getParent()***

Gets the target's parent Node.

Public void ***setParent*** *(<u>Node</u> parent)*

Sets the target's parent Node. The parent may be validated by the child to determine whether or not it is a suitable parent. If it is not, then an IllegalArgumentException must be thrown.

Public void ***addNodeListener****(<u>NodeListener</u> listener)*

Register a<@link metadyne.content.NodeListener> with the target to receive<@link metadyne.content.NodeEvent>s.

Public void ***removeNodeListener*** *(<u>NodeListener</u>) listener)*

Deregister a previously registered<@link metadyne.content.NodeListener>.

Public void ***update()***

Update the target and notify all <u>NodeListeners</u> of a valid change. This method must be called by applications once the value object returned by <u>getValue()</u> has been modified, so that any other objects using the target are notified of the change.

Public void ***acceptVisitor****(<u>NodeVisitor</u> nodeVisitor)*

This method allows a visitor pattern to be applied to the set of trees. To get the back of the tree of nodes in a particular order.

Connections

A "Connection" is a type of Resource that is used as an end-point of some communications link for sending and retrieving data. Connection implementations may either be client side where they are used to access a service or transport facility, or server side where they are used to provide the service or transport facility.

Connections are defined by the Connection API illustrated in the hierarchy diagram of Figure 10. This API defines three new interfaces called Connection, ClientConnection and ConnectionListener and three new classes called ConnectionInfo, ConnectionDiscoveryTemplate and ConnectionEvent. The ClientConnection interface extends the Connection interface; ConnectionInfo extends ResourceInfo; ConnectionDiscoveryTemplate extends ResourceDiscoveryTemplate; ConnectionListener extends the EventListener interface of the Java Utility API and ConnectionEvent extends EventObject of the Java Utility API.

Connections are defined in the preferred system by an interface called Connection, implementations of which represent different communication protocols or services. The Connection interface is an enhancement of Java's URLConnection class, which is intended for the same purpose as Connection, but falls short of providing essential functionality for dynamic applications as described earlier. Connection implementations, however, can be discovered dynamically according to their protocol using the Resource Registry, decodes data using dynamically discovered decoders into appropriate Content objects, encodes Content objects using a dynamically discovered encoder, and is also abstract enough as a definition to support both client and server side implementations. Encoders are implemented as ExternalizingFormats and decoders as InternalizingFormats (see "Formats" later in this document).

The Connection interface defines methods for the same basic operations supported by URLConnection, such as opening and closing a connection, reading and writing bytes through the connection, and querying the type of the data being read. In addition, Connection defines methods not found in URLConnection, for sending and retrieving Content objects with automatic encoding and decoding, querying of connection capabilities such as real-time communication support, and registration of event listeners with a Connection instance for notification of communication status such as termination by the opposite end-point and delayed communication due to heavy network load.

Client side Connection implementations are defined by an interface in the preferred system called ClientConnection, which extends the Connection interface. The ClientConnection interface defines a Connection for use on the client-side of a communications link, i.e. the initiator of the connection. The ClientConnection interface defines a few simple methods for client-side use such as sending requests to a server and receiving responses from the server. These methods are normally used automatically by the connection itself when sending and retrieving data through that connection. Since the Connection and ClientConnection interfaces define the same methods as Java's URLConnection class, together with those for the additional capabilities, it is possible for ClientConnection implementations to subclass URLConnection for compatibility with Java's Networking APIs.

The following table identifies example implementations of the ClientConnection interface currently in development.

Implementation Name	Protocol/Service	Use
HTTPClientConnection	HyperText Transport Protocol (HTTP)	Access to HTTP (World Wide Web) Servers
FTPClientConnection	FileTransfer Protocol	Access to FTP Servers
ClipboardClientConnection	System Clipboard	Cut and paste data transfer
FileClientConnection	Local Filesystem	Local File Access
SMTPClientConnection	Simple Mail Transfer Protocol (SMTP)	Sending e-mail via SMTP Servers
POPClientConnection	Post Office Protocol (POP)	Retrieval of e-mail from POP Servers
SQLClientConnection	SQL Relational Database access protocol	Access to database servers supporting the Structured Query Language

A similar ServerConnection interface may also exist for use with a supporting framework for server side connections which will support retrieving request messages, processing those messages to discover, activate, and use Actors that will service the request and send response messages. Such a ServerConnection and supporting framework are not described in this document.

Connection implementations are described by associated information object classes that must subclass ConnectionInfo - a class defined in the preferred system that extends ResourceInfo for registration with the Resource Registry. For each implementation of the Connection interface or one of its sub-interfaces, there must exist a single subclass of ConnectionInfo, following the requirements of resource discovery. Discovery criteria must be specified within an instance of a class called ConnectionDiscoveryTemplate, a subclass of ResourceDiscoveryTemplate defined in the preferred system.

The information accessible from a ConnectionInfo object and which may be specified as criteria within an instance of ConnectionDiscoveryTemplate consists of the end-point type, protocol or service name, and capabilities such as real-time transfer and random access.

Essentially, only the end-point type and protocol are required to discover a particular Connection implementation. Once an implementation class has been discovered and instantiated, it may be used to send and retrieve data, usually in the form of Content objects.

The following table identifies the essential methods defined by the Connection interface and their purpose.

Method Name	Description
connect	Open the connection.
disconnect	Close the connection.
getContentType	Get the type of the content accessed by the connection as MIME type string.
getInputStream	Get the underlying transfer stream as a Java InputStream object for reading data as bytes.
getOutputStream	Get the underlying transfer stream as a Java OutputStream object for writing data as bytes.
addContentListener	Register an event listener to be notified of connection status.
removeContentListener	Deregister as previously registered event listener.
get	Read the data from the Connection and decode into an appropriate Content object.
put	Encode a given Content object in some format and write the data through the Connection.

Before a Connection instance may be used it must be connected. For ClientConnections, this means that the location of the data to retrieve or to which data is to be sent must be specified to the connection, followed by opening the connection. ServerConnections will need to be given the machine and service address that they are to provide access to, followed by opening the connection. This method and other control methods are most likely invoked by code in the Actor using the Connection.

Once instantiated, the connection must be told which location to use by invoking its 'setURL' method (defined by the ClientConnection interface), and then opened, i.e. an actual connection is made to the specified location, by invoking its 'open' method. The 'open' method must be implemented by the connection to create an underlying network connection to the remote service, or to access a local service such as a hard disk file system.

Data is retrieved through a connection by invoking its 'get' method. The 'get' method must be implemented by the connection to send a request to the server for the data, determine the encoding type of the data known as content type resolution, discovery of a suitable InternalizingFormat to decode the data, setting-up and use of the InternalizingFormat to decode the data, use of the InternalizingFormat to read the data and decode it into a ContentObject, and finally returning that Content object to the caller of the 'get' method.

A request must be sent in a protocol or service specific manner. Content type resolution typically involves checking for the MIME type of the data in the response obtained from the server. If that information is not available then enough of the data should be read to look for a magic number, an identification code often placed towards the beginning of files to identify their type. If the magic number is not available then the fileName suffix, i.e. the characters after the last '.' in a fileName such as ".jpg" for JPEG image files, may be used to determine the type. This information may be used as criteria for discovery of an InternalizingFormat that is able to decode that type of encoded data. The use of the InternalizingFormat is described under "Formats" later in this document.

Data is sent through a connection by passing a Content object as a parameter to the connection's 'put' method. An encoding must also be specified, either as a FormatInfo object that describes an ExternalizingFormat implementation or as a MIME type string, which is used to discover a suitable ExternalizingFormat. The 'put' method must be implemented to discover the ExternalizingFormat if required, configure the ExternalizingFormat to use the connection, send a request to the server to accept the data, and use of the ExternalizingFormat to encode and write the data through the connection. The 'put' method fails if the FormatInfo describes an ExternalizingFormat that cannot encode the specified Content object. The use of the ExternalizingFormat is described under "Formats" later in this document.

Once the connection has fulfilled its usefulness it is closed to release any system resources associated with the Connection, such as network connections or local files. A connection instance is closed by invoking its 'disconnect' method.

Connection implementations are also expected to notify event listeners, registered with the Connection, of the status of the connection as it changes. Listeners must implement the preferred system's ConnectionListener interface in order to monitor a Connection's status. Changes in status are themselves represented by Instances of the ConnectionEvent class. The types of status and reason for occurring are identified in following table.

Status	**Reason**
connecting	The 'connect' method was invoked and the connection is attempting to connect to a service.
connected	Connection to the service has been established.
sending request	A request is being sent to the service such as after invoking the 'get' or 'put' methods.
response received	A response to a request has been received from the service.
retrieving content	Data is being retrieved and decoded.
content retrieved	Data retrieval is complete.
sending content	Data is being encoded and sent.
content sent	Data sending is complete.
transfer failed	Sending or retrieving data failed, probably due to internalization or externalization failure.
connection failed	The connection failed for some reason and will be disconnected. Any pending data transfer and processing will be aborted.
disconnecting	The connection is being closed due to the 'disconnect' method being invoked or a connection failure.
disconnected	The connection has been closed.

The following example code illustrates how a Client Actor for accessing HTTP (Web) servers, embedded within and activated by a hosting Web browser application, retrieves the data from a location specified by the user:

```
// discover a ClientConnection for the HTTP protocol
ConnectionDiscoveryTemplate template =
new ConnectionDiscoveryTemplate("client","http");

ResourceInfo[ ] infos =
ResourceRegistry.lookup(template);

// the first discovered implementation will do
ConnectionInfo info = (ConnectionInfo) infos[0];

// instantiate the connection and set its location
Class connectionClass = info.getResourceClass();
ClientConnection connection =
connectionClass.newInstance();
connection.setURL(userSpecifiedLocation);

// open the connection
connection.connect();

// retrieve the data
Content theData = connection.retrieveContent();

// make the data available to other Actors within the
// browser
appcontext.setAppContextState(theData);
```

With the first line of code the Actor instantiates a new ConnectionDiscoveryTemplate which specifies that the Connection be a Client for the 'http' protocol. The next line invokes the lookup method of the Resource Registry passing the template as a parameter to the method. The results of the Registry search are returned as an array of ContentInfos called infos. The Actor may use any of these to retrieve the data and has been encoded simply to use the first returned Connection in the array of ContentInfos. The next three lives instantiate the Connection from the class information of its ContentInfo and set the designated RLL of the location from which data is to be retrieved. The Connection is then connected and the data retrieved using the Connections retrieveContent or get method. This data is then placed into the AppContext state object. In this example no decoding of the data using a Format was considered.

The following example code illustrates how a Client Actor for accessing HTTP (Web) servers, embedded within and activated by a hosting Web browser application, sends the data from the application to a location specified by the user, encoding it using a Format. The first lines of the code are the same as those above, only in this case the location specified in the SetURL method is the destination location. Once the Connection is instantiated the data needs to be obtained and read out.

```
// get the data from the application
Content theData = (Content)
appcontext.getAppContextState();

// discover available ExternalizerFormats for
// encoding the data
FormatDiscoveryTemplate formatCriteria = new
FormatDiscoveryTemplate(theData.getClass()
.getName());

ResourceInfo[ ] infos =
ResourceRegistry.lookup(formatCriteria);

// invoke an internal method within the Client to ask
//the user
// which encoding is to be used
FormatInfo selectedEncoding =
getEncodingSelectedByUser(infos);

// open the connection
connection.connect();

// send the data
connection.sendContent(theData,selectedEncoding);
```

The first line of the above code gets the data from the current AppContext using its getAppContext state method. The method returns an instance of Content called 'the Data'. The returned values from the getClass and getName methods of the Content are then passed as parameters to the FormatDiscoveryTemplate constructor method for creating a discovery template capable of converting that Format some external data encoding. The template is passed to the Resource Registry's look-up method and an array of FormatInfos are returned. The array contains a FormatInfo for each external encoding of data into which data of the given Content type may be converted by the Formats installed in the Registry. The next line assumes that the Actor requests the user to chase the external data encoding from those available encodings discovered from the Registry, using one of its own internal methods. Once the user has selected the target encoding the connection is connected and the data read out using the connection's put or sendContent method and the appropriate Format.

Connection Interface

public void connect()

Connects the target to the underlying transport service. This method must fire a ConnectionEvent of type CONNECTING, before connecting, followed by firing a ConnectionEvent of type CONNECTED once successfully connected. Failure to connect must result in a ConnectionEvent of type CONNECTION_FAILED.

Public void ***disconnect()***

Disconnects the target from the underlying transport service. This method must fire a ConnectionEvent of type DISCONNECTING, before disconnecting, followed by firing a ConnectionEvent of type DISCONNECTED once disconnected.

This method may be invoked by applications or by the connection itself. A disconnected connection cannot be used, but can be reconnected by invoking connect() or a method that relies on being connected.

Public boolean ***isConnected()***

Returns true if the target is actually connected, i.e. its connect() method has been invoked and disconnect() has not yet been invoked.

Public boolean ***isActivelyTransferring()***

Returns true if the target is actively transferring data, i.e. a send or retrieval operation is pending

Public int ***getContentLength()***

Public ExternalContentDescriptor ***getExternalContentDescriptor()***

Returns information about the external content that is retrievable via the target or null if no such information is available.

Public java.io.InputStream ***getInputStream()***

Returns an InputStream for reading from the target. The stream must implement the mark() and reset() methods defined by java.io.InputStream.

Public java.io.OutputStream ***getOutputSteam()***

Returns an OutputStream for writing to the target.

Public void ***addConnectionListener*** *(<u>ConnectionListener</u> ccl)*

Registers the specified event listener to be notified of ConnectionEvents.

Public void ***removeConnectionListener*** *(<u>ConnectionListener</u> ccl)*

Deregisters the specified event listener.

Public <u>Content</u> ***retrieveContent()***

Retrieves the external data that the Connection has access to, or throws an IOException if no data is available to retrieve.

This method must firstly connect(), followed by sending a request for retrieval according to the implementation specific protocol. Once the request is sent, the target musts first a ConnectionEvent of type SENDING_REQUEST. Once a response has been received from the opposite end-point, a ConnectionEvent of type RESPONSE_RECEIVED must be filed.

The target must then fire a ConnectionEvent of type RETRIEVING_CONTENT, before resolving the external format/encoding of the data and using that information to discover and use an InternalizerFormat to internalize the data into a Content object. Once the Content object has been obtained from the InternalizerFormat, the target must fire a ConnectionEvent of type CONTENT_RETRIEVED and return the Content object.

An IO failure within the target itself must result in a ConnectionEvent of type CONNECTION_FAILED being fired, while a failure within the InternalizerFormat must result in a ConnectionEvent of type TRANSFER_FAILED being fired. In both cases, an IOException must be thrown and retrieval aborted.

Public <u>Content</u> **retrieveContent** (<u>PropertyStore</u> intParamas, <u>FormatListener</u> listener)

*Public <u>Content</u> **retrieveContent** (<u>FormatListener</u> listener)*

Retrieves the external data that the Connection has access to, or throws an IOException if no data is available to retrieve.

*Public void **sendContent** (<u>Content</u> content, <u>PropertyStore</u> externParams, (<u>FormatListener</u> listener)*

Sends the specified Content through the target to the opposite end-point, encoded according to the MIME type specified by the Content's "currentExternalFormat" property.W.

This method must firstly connect(), followed by discovering a suitable ExternalizerFormat implementation to externalize the data using the "currentExternalFormat" MIME type as the discovery criteria. If content==null or no InternalizerFormat was discovered, then a ConnectionEvent of type

TRANSFER_FAILED must be fired, followed by throwing an IOException.

Once a suitable ExternalizerFormat has been discovered and instantiated, the target must send a request for sending data according to the implementation specific protocol, to the opposite end-point. Once the request is sent, the target must fire a ConnectionEvent of type SENDING_REQUEST. Once a response has been received from the opposite end-point, a ConnectionEvent of type RESPONSE_RECEIVED must be fired.

The target must then fire a ConnectionEvent of type SENDING_CONTENT and then use the ExternalizerFormat to encode and write the data, passing it the Map of externalization parameters passed to this method. The externalization parameters enable an application to set encoding options or control flags, such as compression ratios. Once externalization is complete, the target must fire a ConnectionEvent of type CONTENT_SENT.

An IO failure within the target itself must result in a ConnectionEvent of type CONNECTION_FAILED being fired, while a failure within the ExternalizerFormat must result in a ConnectionEvent of type TRANSFER_FAILED being fired. In both cases, an IOException must be thrown and sending aborted.

Applications may register for the ConnectionEvents via the addConnectionListener() method. Further information in terms of externalization of the internal Content data for writing through Connections may be obtained as FormatEvents from the ExternalizerFormat. Since applications do not have direct access to the ExternalizerFormat, they may only register by passing a FormatListener to this method. See Format and ExternalizerFormat for details of when FormatEvents are fired.

*Public void **sendContent** (<u>Content</u> content, java.lang.String mimeType, <u>PropertyStore</u> externParams, <u>FormatListener</u> listener)*

ClientConnection Interface

*Public <u>URI</u> **getURI()***

Returns the Uniform Resource Identifier that the target is bound to.

*Public void **setURI** (<u>URI</u> uri)*

Set the Uniform Resource Identifier (URI) with which the target is to bind. This method may only be invoked before connect(), otherwise an IllegalStateException is thrown.

*Public void **setURI** (java.lang.String uriAsString)*

Set the Uniform Resource Identifier (URI) with which the target is to bind. This method may only be invoked before connect(), otherwise an IllegalStateException is thrown.

*Public java.langString **getResourceName()***

Returns the name (atomic or composite) of the resource referenced by the target's URI.

*Public void **setResourceName** (java.lang.String name)*

Sets the name (atomic or composite) of the resource to be referenced by the target's URI. The specified name replaces the name part of the target's URI for the next request.

This method may be invoked only if a URI has been set, either via a constructor or setURI(), and the target is not actively transferring data.

Public void ***setRequestProperties*** *(PropertyStore req)*

Sets the parameters of the next request to be sent. A request is sent for operations that need to request action by a server or transfer mechanisms. Some connection protocols, such as HTTP, allow arbitrary request data to be sent to servers. This method allows an application to specify protocol specific or location and application specific request data which is combined with request data generated by the target for the next operation that requires sending a request.

The request data is specified as key-value pairs which musts be sent according to the underlying protocol. Any key-value pair that is not supported by the target must be ignored. If the target does not support setting of properties at all, then this method must ignore the parameter passed to it.

This method may not be invoked while sending or retrieval of content is pending.

Public PropertyStore ***getRequestProperties()***

Returns a PropertyStore object that may be used to get and set request parameters, or null if request parameters cannot be set externally to the target.

CleintConnection implementations may send request parameters as part of their protocol, but some may allow applications to set additional parameters, such as HTTP header fields. This method may be used to retrieve an object into which additional parameters may be added and any already set by the target or previously by the application can be examined.

Implementations must ensure that modification of the request parameters during an active request is not permitted, or does not affect the integrity of the request. Synchronization of the properties or use of a copy of the properties could be used, for example.

Public PropertyStore ***getResponseProperties()***

Returns a PropertyStore object that may be used to get the response parameters received by the target for a previously sent request. The return value may be null if no request has been sent since the target was last connected, or if the target does not use response parameters.

Public void ***makeDirectory*** *(java.lang.String name)*

Creates a directory or namespace relative to a directory or namespace identified by the target's URI. The URI must identify a directory or namespace into which the new directory or namespace may be nested.

Public java.lang.String[ ] ***list()***

Returns the canonical names of the resources that are accessible relative to the target's URI. The URI must be a directory or namespace.

Public java.lang.String[ ] ***list****(java.io.Filenamefilter filter)*

Returns the canonical names of the resources that are accessible relative to the target's URI. The URI musts be a directory or namespace. If a java.io.FileName filter is specified, then the names are passed to its accept() method to determine whether or not each name should be included in the result.

Public void ***delete()***

Deletes the resource referenced by the target's URI.

Public void ***rename*** *(java.lang.String newName)*

Changes the name (the right most part if a composite name) to the specified name and updates the target's URI to reflect the new name.

ConnectionInfo Class

The items of information available through this class are as follows:

name

the human readable name of the connection, suitable for a menu item.

shortDescription

a short human readable description, suitable for a tooltip

longDescription

a longer human readable description, suitable for an "About" dialog

manufacturerName

the name of the Connection's developer or manufacturer

connectionClassName

the fully qualified name of the Connection subclass

customizerClassName

the fully qualified name of the Connection customizer component

copyright

the Connection object's copyright string *version*

the Connection object's version number, expressed as an integer

majorRevision

the Connection object's major revision number, expressed as an integer

protocolName

the human-readable name of the protocol supported by this Connection

realtime

indicates whether isochronous delivery is supported by this Connection

reliable

indicates whether reliable delivery is supported by this Connection

streaming

indicates whether the protocol itself is streaming rather than random access

pushDelivery

indicates whether this Connection supports push delivery

pullDelivery

indicates whether this Connection supports pull delivery

role

the String "client" or "server" indicating which role it plays in an connection *requestPropertyDescriptors*

RequestPropertyDescriptor for each request property supported by the connection.

The class defines methods for accessing all of these items of information as well as get LookupAttributes and get ResourceType methods.

ConnectionDiscoveryTemplate class

This class, an extension to the ResourceDiscoveryTemplate class, defines fields that correspond to information held within ConnectionInfo implementations. The following fields are defined by this class:

The class defines three methods, getLookupAttributes, getResourceType and filterLookup Results which override those of ResourceDiscoveryTemplate with corresponding name.

Formats

A Format is a type of Resource, which encodes Contents for writing through a Connection to some location, or decode data read through a Connection from some location into a Content.

Formats are defined in the preferred system by the interfaces and classes of the Format API, illustrated in the hierarchy diagram of Figure 11. The API defines three new interfaces, these being an interface called Format and two sub interfaces of that interface called InternalizerFormat and ExternalizerFormat. Two classes called FormatInfo and FormatDiscoveryTemplate are defined which extend the corresponding classes of the ResourceRegistry API so that implementations of Formats may be discovered from the Registry.

Formats are typically requested and controlled by a connection to decode or encode the data being transferred by the connection. The Format involved in the transfer is typically contained within the controlling connection.

When a Connection is used to retrieve data, it determines the type of the data and then uses the Resource Registry to discover the appropriate Format implementation to decode that type of data. The Format is then used to read the data from the Connection and decode the data into a suitable

Content implementation for the data type, e.g. a Format implementation for decoding JPEG images would decode the data into an instance of ImageContent. The process of reading, decoding, and encapsulating the data into a Content object is known as "internalization".

When a Connection is used to send data, the entity using the Connection (such as an Actor), must specify the required encoding either as a FormatInfo that describes the specific Format implementation or a MIME type that is used to discover a suitable Format implementation. The specified or discovered Format implementation is then used to get the data from the Content object to be sent, encodes that data, and then writes it through the Connection. The process of obtaining getting the Content data, encoding it, and writing the encoded data through a Connection is known as "externalization".

If the entity using the Connection does not specify the external encoding of the data then the Connection may invoke the getExternalFormat method of the Content being externalized to determine the default data encoding for the Content. The appropriate Format may then be requested from the Resource Registry.

An implementation of the Format interface is either an encoder ("externalizer") or decoder ("internalizer") for a particular encoding format. Implementations do not implement the Format interface directly, but instead must implement one of two interfaces that extend the Format interface. The two interfaces are defined in the preferred system as InternalizerFormat and ExternalizerFormat. Implementations of the former are internalizers or decoders, while implementations of the latter are externalizers or encoders. An implementation of the InternalizerFormat is required for each encoding format to be supported when retrieving data through a Connection and an implementation of ExternalizerFormat is required for each encoding format to be supported when sending data through a Connection. Every implementation is associated with a single encoding format only and a single Content implementation. For example, an implementation of InternalizerFormat for JPEG encoded images decodes only JPEG image data and consistently create an ImageContent object to encapsulate the decoded data.

Example encoding formats and their corresponding Content implementations are shown in the following table.

Encoding Format	Description	Associated Content Implementation
JPEG	JPEG Image Format	ImageContent
GIF	GIF Image Format	ImageContent
HTML	HyperText Markup Language	DocumentContent
RTF	Rich Text Format	TextContent

Implementations of InternalizerFormat and ExternalizerFormat, collectively called 'Formats', are described by associated subclasses of a class called FormatInfo, which is a subclass of the ResourceInfo class. Implementations of both InternalizerFormat and ExternalizerFormat may be registered for discovery using the Resource Registry by specifying discovery criteria within an instance of a class called FormatDiscoveryTemplate, defined in the preferred system and a subclass of the ResourceDiscoveryTemplate class.

The description of a Format that may be obtained from its associated FormatInfo and the criteria by which Formats may be discovered consists of a MIME type for the encoding format, a "magic number" value, associated fileName suffixes, and the associated Content implementation.

The methods defined by the Format interface are shown in the table below.

Method Name	Description
setConnection	specify the Connection instance to use for internalization or externalization
addFormatListener	register an event listener to be notified of the status of internalization or externalization
removeFormatListener	deregister a previously registered event listener
dispose	terminate internalization or externalization and free up the system resources used by the process

In addition, the InternalizerFormat interface defines a method called 'internalize' and the ExternalizerFormat interface defines a method called 'externalize'. Implementations of InternalizerFormat must implement the 'internalize' method and the methods defined by the Format interface. Implementations of ExternalizerFormat must implement the 'externalize' method and the methods defined by the Format interface.

A Connection uses a Format by firstly passing a reference to itself as the parameter to the Format's 'setConnection' method. The Connection then invokes the format's 'internalize' or 'externalize' method to actually perform the internalization or externalization process. If the Connection's 'dispose' method is invoked while it is using a Format, then it invokes the Format's 'dispose' method to abort the internalization or externalization process..

The 'internalize' method must be implemented to read the data from the Connection (commonly using a Java InputStream object obtained from the Connection's 'getInputStream' method), create an instance of the appropriate Content implementation for the data, decode the data into an internal representation suitable for encapsulation within the Content, put the internal representation into the Content object, set the Content object's 'current external format' to the implementation that just decoded the data, and return the Content to the caller of the 'internalize' method. If the InternalizerFormat is disposed of while internalising then it must stop internalising, dispose of the Content object, and fail.

If the data being internalized contains some other data of a different encoding, then the InternalizerFormat that is internalising the enclosing data needs to delegate decoding of the embedded data to another suitable

InternalizerFormat. The encoding format of the embedded data must be determined first, followed by using the Resource Registry to discover a suitable implementation of InternalizerFormat. The discovered implementation is then used like any other to internalize the embedded data into a Content object. The InternalizerFormat of the enclosing data, then places that Content object into the Content object that represents the enclosing data.

The 'externalize' method must be implemented to get the data from the Content, encode that data into the external representation, and write that encoded data through the Connection (commonly using a Java OutputStream object obtained from the Connection's 'getOutputStream' method). Upon successful completion of the externalization process, the Content object's 'current external format' must be set to the implementation that just encoded the data. If the ExternalizerFormat is disposed of while externalising then it must stop externalising and fail.

If the Content being externalized contains some other Content, then the ExternalizerFormat must delegate encoding of the embedded Content to another suitable ExternalizerFormat. The type of encoding used for the embedded Content is determined in an implementation specific manner, such as being fixed according to the capabilities of the encoding used for the enclosing Content, specified within the enclosing Content in some implementation specific manner, the 'current external format' of the embedded Content may be examined, or the first encoding discovered that is able to encode the embedded Content may be used. The determined encoding is used to discover the associated ExternalizerFormat implementation, which is used like any other to externalize and write the embedded Content through the Connection.

InternalizerFormat and ExternalizerFormat implementations must notify any registered event listeners of internalization and externalization status. The types of status and reason for occurring are identified in following table.

Status	Reason
processing started	Internalization or externalization has started, due to the 'internalize' or 'externalize' method being invoked.
waiting for data	The Format is waiting for data to be retrieved by the Connection, for externalization. This may occur during an unexpected delay in data transfer.
validating	The data is being validated to ensure it is encoded correctly and/or the data structure is not corrupt or malformed.
validated	Validation was successful.
validation failed	Validation has failed for some reason, processing will not continue.
processing complete	Internalization or externalization has finished successfully.
processing failed	Internalization or externalization has failed for some reason, perhaps due to corrupt data or a connection failure.
processing aborted	Internalization or externalization has stopped due to disposal of the Format during the process.
disposed	The Format instance has been disposed of.

The following example code illustrates how the 'get' method could be implemented by a Connection to internalize data read from an instance of that Connection (referred to as 'this') into a Content object.

```
public Content get() {

// resolve the encoding format of the data at the
// Connection's location, using an implementation
//specific method
String mimeType = getMimeTypeOfDataEncoding();

// discover an InternalizerFormat for the encoding
// type
FormatDiscoveryTemplate template =
new FormatDiscoveryTemplate("internalizer",mimeType);

ResourceInfo[ ] infos =    Resource
Registry.lookup(template);

// any matching implementation can be used, so if
// more than one was found then just use the first
// one in the result array
FormatInfo formatInfo = (FormatInfo) infos[0];

// instantiate and set-up the InternalizerFormat
Class formatClass = formatInfo.getResourceClass();

InternalizerFormat format =(InternalizerFormat)
formatClass.newInstance();

format.setConnection(this);

// read the data from the Connection and internalize
// into Content to be returned to the caller of this
// method
Content data = format.internalize();
return data;
}
```

The first line of the code determines the MIME type of the external data encoding to be internalized, returning this information as a String called mimeType. The next line of the method code uses the identified MIME type as a parameter to pass to the constructor method of the FormatDiscoveryTemplate. The other parameter is 'internalizer' since data is to be read in from an external encoding to a Content. The discovery template is then passed to the ResourceRegistry's lookup method and an array of suitable FormatInfos returned in an array called infos. Since any matching implementation of a Format may be used in this case, the first matching

Format, that at index zero in the infos array, is chosen to be instantiated from its FormatInfo. The next three lines get the Format's class from the FormatInfo, instantiate it, and sets the connection to that which requested the Format from the Registry. The last line uses the 'internalize' method of the Format to write the data into a Content object, and return that Content.

The following example code illustrates how the 'put' method could be implemented by a Connection to externalize a Content object through an instance of that Connection.

```
public void put(Content data, String mimeType) {

// if a MIME type is not specified, use the
// Content's current external format'
if(mimeType==null) {
     mimeType = data.getCurrentExternalFormat();
     }

// discover an ExternalizerFormat to encode in the
//format specified by the MIME type
FormatDiscoveryTemplate template =
new FormatDiscoveryTemplate("externalizer",mimeType);

ResourceInfo[ ] infos =    Resource
Registry.lookup(template);

// any matching implementation can be used, so if
// more than one was found then just use the first
// one in the result array
FormatInfo formatInfo = (FormatInfo) infos[0];

// instantiate and set-up the ExternalizerFormat
Class formatClass = formatInfo.getResourceClass();

ExternalizerFormat format =(ExternalizerFormat)
formatClass.newInstance();

format.setConnection(this);

// externalize the data and write through the
Connection
format.externalize(data);
}
```

The put method of the Connection may take two parameters, the Content to be encoded, and the MIME type of the data format into which it is to be encoded.

The first line of the method is executed if no MIME type is passed to the method, and invokes the getCurrentExternalFormat method of the Content being externalized in order to resolve a MIME type for encoding. Once a MIME type is known the necessary ExternalizerFormat can be requested from the Registry, instantiated and set to work with the requesting connection.

The Format's externalize method is finally invoked to read the Content data out through the Connection and write it into the specified external data type.

Format Interface

public void ***setConnection*** *(connection)*

Set the connection that the target is to use to read data for internalization or write data that has been externalized.

public *Connection* ***getConnection()***

Get the connection that the target is using to read or write data.

public void ***addFormatListener*** *(FormatListener)*

Registers the specified listener to be notified of the progress of content externalization/internalization.

public void ***removeFormatListener*** *(FormatListener)*

Deregisters the specified listener.

public void ***dispose()***

Disposes of the resources held by the target including release of FormatListeners and the Connection.

public Content internalizeContent (PropertyStore externalizeParameters)

Reads data from the target's current connection, and decodes (internalizes) that data into a content object.

The ExternalizerFormat interface

public void externalizeContent (Content, Property Store, externalizeParameters)

retrieves data from the specified Content and encodes (externalizes it, before writing the encoded data via the target's current connection.

The FormatInfo class

The following items of information are provide by this class:

name

the human readable name of the Format suitable for a menu item

shortDescription

a short human readable description suitable for a tool-tip

longDescription

a longer human readable description suitable for an "About" dialog

manufacturerName

the name of the Format's developer or manufacturer

formatClassName

the fully qualified name of the Format subclass *customizerClassName*

the fully qualified name of the Format customizer component

copyright

the Formats copyright string

version

the Formats version number, expressed as an integer

majorRevision

the Formats minor revision number, expressed as an integer

mimeType

the MIME type supported by the FormatcontentClassName - the fully qualified name of the Content subclass into which the Format may internalize data from a Connection

magicNumber

the header value that identifies data externalized in the content type handled by the Format

magicNumberOffset

the number of bytes from the start of data, where the magic number is located

fileTypeSuffixes

the file type suffixes that may be added to filenames of data externalized in the content type handled by the Format.

The methods defined by this interface are defined as follows:

public final int ***getFormatType()***

get the type of the format as an integer matching one of the constants defined by FormatTypeConstants *public final MimeType[ ]* ***getMimeTypes()***

returns the MimeTypes associated with this Format

public final java.lang.String[ ] ***getMimeTypesAsStrings()***

returns the MimeTypes associated with this Format as a human-readable String

public final java.lang.String ***getContentClassName()***

returns the class name of the Content associated with this Format

public final java.lang.Class ***getContentClass()***

returns the Class of the content associated with this format

public final byte[ ] ***getMagicNumber()***

returns the magic number associated with this Format

public final int **getMagicNumberOffset**()

get the offset of the magic number associated with this Format

public final java.lang.String[ ] ***getFileTypeSuffixes()***

returns the file name suffix(es) associated with this Format

public java.lang.Class ***getResourceType()***

Overrides:

getResourceType in class ResourceInfo

See Also:

ResourceInfo.getResourceType()

public java.util.Map ***getLookupAttributes()***

Overrides:

getLookupAttributes in class ResourceInfo

See Also:

ResourceInfo.getLookupAttributes()

public java.lang.String ***toString()***

Overrides:

toString in class java.lang.Object

See Also:

Object.toString()

protected static *MimeType*[ ]

createMimeTypes*(java.lang.String*[ ] mimeTypesAsStrings)

FormatDiscoveryTemplate class

This class, an extension to the ResourceDiscoveryTemplate class, defines fields that correspond to information held within FormatInfo implementations.

The FormatDiscoveryTemplate class defines three methods called getResourceType, getLookupAttributes, and filterLookupResults. These methods override the methods in ResourceDiscoveryTemplate with corresponding name. The getResourceType method necessarily returns 'Format' for Format implementations.

Example Implementations

Figure 12 is a hierarchy diagram which illustrates how example implementations of each of the four types of Resource utilize the classes and interfaces of the preferred system.

An ImageViewer Actor is shown implementing the Actor interface, and extending the Object class of the Java language core API. An ImageViewerBeanInfo is provided which extends the ActorInfo class, and provides information about the viewer in addition to that specified in the ActorInfo.

Similarly an HTTP ClientConnection implementation is shown which implements the ClientConnection interface. A corresponding subclass of ConnectionInfo called HTTPClientConnectionBeanInfo is also shown. An ImageContent implementation of the Content interface with its corresponding subclass of ContentInfo, called ImageContentBeanInfo, and a JPEGInternalizerFormat implementation of the InternalizerFormat interface, and JPEGExternalizerFormat implementation of the ExternalizerFormat interface, with their corresponding subclass implementations of FormatInfo are also shown.

Each of the implementations of the Resource interfaces provides the method code to perform the methods defined in the interfaces which they implement. Similarly, each of the BeanInfo subclasses of the different ResourceInfo subclasses, provides the information defined in those subclasses, so that the Resources may be discovered in the Registry and any additional information unique to that Resource.

Specific Examples

It will be appreciated from the foregoing general discussion that the invention is large in scope. To better illustrate how the components and Resources operate according to the preferred system at runtime, two specific examples of use will now be given.

The first example illustrates the use of a web-browser provided with adaptive functionality to create and send an email message. The second example illustrates the use of a web server to obtain and store share price information.

Example 1

In the first example of use of the preferred system a user will create an email, attach an image file obtained from a file, and send the email to an addressee. The operations that are executed by the preferred system during use will be illustrated with reference particular reference to Figures 13 to 20.

Figure 13 shows a web browser component 100 provided on a computer running the Java Virtual Machine and into which the APIs of the Java platform and of the preferred system have been installed. The browser acts as the root container in which the dynamic application is to be constructed. The browser is launched and initialised, either automatically when the computer is started up, or as is more likely, in response to a user selecting an icon or menu option for the browser presented on the default GUI of the computer operating system, the 'desktop' for example.

In the start up file of the Web Browser are commands to instantiate an instance of the ResourceRegistry class, and configure this for use with an external implementation of the ResourceRegistryProvider interface in which the Resources available to the Web Browser are stored. The instance of the ResourceRegistry may then be accessed by the Web Browser and by any Resources it contains in order to request the Resources which provide the desired functionality. The Registry is represented in Figure 13 by the ResourceRegistry 200.

The browser's start up code also contains a request for an empty AppContext:

```
AppContext anAppContext = new AppContext();

DefaultAppContextProvider provider =
new DefaultAppContextProvider(anAppContext);

thisContainer.addservice(AppContext.class, provider)
```

Following execution of this code a default AppContext 102 is instantiated and contained within browser 100. AppContext 102 contains Attributes 104 but no State object as no data has yet been read into the browser. The browser registers via the RSCP defined in Java's BeanContext API as an event listener of the Attribute contained in the AppContext.

The browser component presents the user with a number of general control options via a Graphical User Interface (GUI) which may take the form of a menu system or a number of icons or buttons. The options that the browser component presents are implementation specific, that is they are specified in the code which defines the browser component.

One of the control options may be an 'Open Content' option for opening, that is retrieving and displaying data held in a file. In response to a user selecting this option, the browser is caused to execute its method for opening a file.

This method first causes an ActorDiscoveryTemplate to be instantiated with the RoleDescriptor specified as 'Client'. The 'Client' Role identifies the Actor as one which provides means to navigate a service, such as a file system, to obtain data.

```
ActorDiscoveryTemplate template =
new ActorDiscoveryTemplate();

template.roleName = "client"
```

Next, the method passes the instantiated ActorDiscoveryTemplate as a parameter to the designated ResourceRegistry's look-up method. The results of this method are returned as an array of ResourceInfos called infoa. The method selects the first returned Info object from the array

```
ResourceInfo infoa =
ResourceRegistry.lookup(template)
```

Next, the browser method returns the human readable names of the Resources returned in the array, using the getResourceName method and presents these to the user in a menu.

A'FileClient' Resource, which provides access to a filesystem on the disk or disks connected to the user's computer, may be found in the look-up and presented to the user in the menu. In response to the user selecting the FileClient option from the browser menu, the browser obtains the class reference of the FileClient Resource from the ResourceInfo, and instantiates, contains and activates FileClient 108. Reference will now be made to Figure 14.

During its activation, the start up code of the FileClient 108 is executed. The FileClient requests the AppContext service from its container, the browser 100, and as a result receives a reference to the browser's AppContext 102.

During activation the FileClient also requests a client-side Connection from the ResourceRegistry for accessing the filesystem using the following ConnectionDiscoveryTemplate.

```
ConnectionDiscoveryTemplate template =
new ConnectionDiscoveryTemplate("file","client");
```

The FileClient instantiates FileClientConnection 112 from the array of suitable ConnectionsInfos returned from the Registry and contains it. The

FileClient now completes its activation and command passes back to the browser's Open Content method.

The browser method requests that the FileClient 110 performs its 'get' operation, which in the case of the FileClient, is an 'open file' operation. In response, the FileClient operation displays a GUI to the user for navigating through the filesystem and selecting a file to open.

By means of the GUI the user locates and selects a file which contains a JPEG encoded image. The FileClientConnection method next attempts to determine the type of the data to read in. This may be achieved by checking the MIME type of the data in the response obtained from the filesystem server, or if that information is not available, by reading in a chunk of the data file in order to determine its type from its Magic number. The data type of the file may also be determined from its file ending. Once the data encoding of the target file is known the Connection may request the appropriate Internalizer Format from the Registry for decoding. A JPEGFormat 116 is requested, instantiated from one of the FormatInfos returned by the Registry and contained within the FileClientConnection 112. The File

The FileClientConnection 112 invokes the internalise method of the JPEGFormat 114, which creates an instance of Image Content, reads the data from the file into the Image Content object , and then invokes the setAppContextState method of the AppContext 102 to place the data into the AppContext. The JPEGFormat knows which AppContext to use by invoking the getService method of its container, the FileClientConnection, specifying 'AppContext' as the parameter of that method. The getService request is passed on by the Connection to its container the FileClient Actor which returns a reference to the AppContext 102 provided to it by the Web Browser.

The JPEGInternaliserFormat, FileClientConnction and FileClient methods completes, the FileClient issues a PerformedOperationEvent to the AppContext, and control passes back to the browser method, which calls the dispose method of the FileClient. As a result the FileClient and the Resources it contains are disposed of.

Reference will next be made to Figure 15. The browser is a registered change listener with the AppContext 102, so that when the new ImageContent 118 is placed into the AppContext, the browser is notified. In response, the browser requests from the ResourceRegistry an Resource which can present image data, namely an Actor with the 'Presenter' Role. A suitable'Viewer' is found and ImageContentViewer 122 is instantiated, contained within the browser and activated. During activation the ImageContentViewer requests an AppContext from the browser using the getService method and receives a reference to AppContext 102. The ImageContentViewer invokes the getAppContextState method of the AppContext and obtains a reference to the ImageContent encapsulated by it. The 'Viewer' displays the ImageContent to the user and issues a performedOperationEvent to the AppContext. The Open Content method of the browser now completes.

The user now views the image data and may check the contents of the selected file. The user decides that they will attach the image file to an email that they will create and send to a recipient.

Reference will now be made to Figure 16. The user selects another option from the browser's menu options, called 'Send Content'. The Send Content method of the browser 100 requests from the ResourceRegistry, an Actor with the 'Client' role. Information about all matching Actors is returned from the Registry in an array of ResouceInfos, which the Send Content method uses to present a list of available 'Clients' to the user. We assume a 'MailClient' Actor is available, which enables the sending and receiving of email messages, and selected by the user. The browser method instantiates, contains and activates the selected MailClient 130 and the MailClient receives access to the AppContext 102.

Reference will now be made to Figure 17. During activation the MailClient 130 creates a new instance 138 of an AppContext with Attributes 140, and using the Registry, a new instance of TextContent 142. The MailClient encapsulates the TextContent in AppContext 138 using its setAppContextState method. The TextContent will be used to receive the body of the user's text message. The MailClient also requests an Actor which has the 'Editor' Role and the discriminator of 'Text' from the ResourceRegistry. This Editor will be used to create the text to form the body of the email message. A TextContentEditor is 'discovered' by the MailClient, instantiated and contained.

During activation The TextContentEditor obtains the AppContext provided by the MailClient using the getService method, registers as an event listener with the Attributes object contained in the AppContext 138 and as a Content Change listener with the TextContent 144. Once activated the TextContentEditor 136 obtains and displays the TextContent data 142 of the AppContext. No text is actually displayed at this stage because the user has not yet entered any data and the TextContent 144 is empty.

During activation the TextContentEditor requests from the ResourceRegistry all Actors with the capability to manipulate TextContent, using an ActorDiscoveryTemplate with the Role 'Tool', and the discriminator 'TextContent'. The MailClient 130 and the Resources it contains are now fully activated for use.

The browser method next 100 invokes the MailClient's 'put' operation, used to send messages, causing the MailClient to display a GUI to the user for entering the recipient's email address, the subject, and other addressing information and causing the Mail Client to invoke the TextContentEditor's 'edit' operation. Upon invoking this method, the TextContentEditor responds by displaying a blank area for entering text and a toolbar containing the icons of all Tools discovered from the Registry or alternatively their text names. The icons, or text names, are obtained using the methods of the ActorInfos retrieved from the ResourceRegistry during activation of the TextEditor.

The user may now enter text and create the email message. As text is entered, the TextEditor stores it in the TextContent 142 encapsulated by the AppContext 138. Reference will now be made to Figure 18.

Having entered the text of the message, the recipient's address, and message subject, the user selects a SpellCheckerTool from the tool bar of the TextContentEditor's GUI. The TextContentEditor 136 instantiates the SpellCheckerTool 148 from the ActorInfos it discovered and contains and activates with access to the AppContext 138.

The TextContentEditor 136 invokes the SpellCheckerTool's 'modify' operation causing it to scan through the TextContent 142 and suggest alterations to words which appear to contain errors. If the user chooses to make changes the SpellCheckerTool 148 modifies the data within TextContent 142 to make those changes to the text. Changes to the text result in an event being created and sent to all event listeners registered with the AppContext. In this case, this will only be the TextContentEditor 136, which responds to such notification by updating the text that is displayed to the user.

Once the spell-checking operation is complete, the 'modify' operation completes and issues a PerformedOperation to Attribute 140 of the AppContext 138. As a registered event listener, the TextContentEditor 136 receives the notification and invokes the dispose method of the SpellCheckerTool 148. Reference will now be made to Figure 19.

The mail message is now ready to send. The user selects the 'Send' option provided on the MailClient's GUI; the Mail Client instantiates a new MessageContent 154 to contain the text and the image of the email message, in a single representation, and deactivates the TextContentEditor 136 since this is no longer needed. The MailClient places the TextContent 142, contained in the AppContext 138 it provided into the new instance of MessageContent 154, and also places the ImageContent 118 data contained within the AppContext 102 it received from the browser 100 into the MessageContent.

The Mail Client next requests from the ResourceRegistry a client-side Connection which is able to send email. The SMTP (Standard Mail Transfer Protocol) ClientConnection 162 is returned by the Registry and instantiated and contained by the MailClient. The MailClient uses the SMTPClientConnection to determine and instantiate the necessary Format connection required to encode the Message content data 154 into the external data format of the email server, which in this case is RFC822. The Connection accordingly requests the appropriate Format from the Registry and causes RFC822Format 164 to be is instantiated and contained. The SMTPClientConnection uses RFC822Format 164 to encode the MessageContent 154, and transmit it to the destination email server.

Once the message has been sent, the MailClient's 'put' operation completes, and the browser 100 disposes of the MailClient 130. When MailClient 130 is removed all the Resources it contains are also removed, leaving only ImagePresenter 122 active in the browser. The end state of the process is shown in Figure 20. The 'Send Content' method of the browser completes.

Example 2

In the second example, the operation of the preferred system is illustrated through use of a Web server which supports dynamic applications. The Web Server will be used to obtain live stock quotes and maintain an historical stock quote database and will be described with reference to Figures 21 to 3g.

Figure 21 shows the root container for the dynamic application in application server 200 in an initial state. The root container in this case is application server 200 which works in the background to maintain stock information and so does not in fact present a GUI to a user to receive commands. The application server may be launched from a menu option or an icon, such as on the desktop. The application server may be interacted with by a browser 310 which provides instructions from a user.

At start-up, the application server requests, activates and contains HTTPServant 202 which is an Actor used to receive instructions from browser 310 using the Hypertext Transport Protocol (HTTP). The request for this component may be set in the configuration files or may be fixed in specialized servers.

The application server also instantiates an instance of ResourceRegistry and configure this to use a pre-determined ResourceRegistryProvider in the manner described earlier. The Resource Registry is represented in Figure 21 by ResourceRegistry 300.

HTTP Servant receives requests from browser 300. The user inputs a request for a Stock Quote Home Page into the browser 310, and the request subsequently arrives at the HTTPServant 202. The home page is a file stored on disk and so will require no further processing. The HTTPServant 202 loads the home page file and sends it to the requesting Web Browser 310 in step S228. The user of the browser views the web page which offers information on stock prices and a number of different methods of accessing that information. One of the options provided by the web page is a Quote History service which tracks the movements of stock prices over time. The user selects this service from the options provided by the web page and causes a request for the QuoteHistoryCapture Actor to be sent through the HTTPServant to the ResourceRegistry of the application server. The

QuoteHistoryCapture Actor may be provided by the supplier of the source of the stock data or the web page developer in order to act as a front end module for accessing the stock data source and returning the information to the viewer of the browser.

The QuoteHistoryCapture 204 Actor is requested, instantiated and contained within the application server to manage the retrieval of the stock information. During initialisation QuoteHistoryCapture requests an AppContext from its container and so receives the default AppContext 206 which contains Attributes 208. QuoteHistoryCapture is a component which runs in the background to update stock information data held in a database. The URL address of the source from which the data is to be obtained and the protocol which should be used to retrieve the data are specified in the code of the QuoteHistoryCapture. The database address in which the quote history information is to be stored is contained within the implementing code of the QuoteHistoryCapture.

Also during its activation QuoteHistoryCapture requests a from the ResourceRegistry, a Connection of the type it needs to access the source of the quote history information. This information is held on the server 320 of a News agency and QuoteHistoryCapture requests an HTTP Client Connection from the Registry in order to access it.

Referring now to Figure 22, the QuoteHistoryCapture 204 uses the information retrieved from the Registry to instantiate HTTPClientConnection 212. The HTTPClientConnection is contained within QuoteHistoryCapture 204 and activated. During activation the HTTPClientConnection determines the format of the data it is to connect to and in step S216 requests a suitable Format from the ResourceRegistry. The Format is not instantiated and contained within the Connection in the usual way, but is not shown in the Figure 22 for convenience. HTTPClientConnection also creates a new TableContent 220 to store the data retrieved from the news agency.

The QuoteHistoryCapture 204 also requests from the ResourceRegistry, in step 218, a client-side connection for SQL (Standard Query Language) access to the database server 330 specified in the code of the component for storing the stock information. SQLClientConnection 222 is accordingly instantiated, contained and activated within the

QuoteHistoryCapture 204. During activation, SQLClientConnection determines the data format of the Database server 330 and requests and instantiates the appropriate Format for encoding. A SQLFormat is returned, instantiated and contained within the connection to handle the encoding but is not shown in Figure 22 for convenience.

The QuoteHistoryCapture 204 is implemented to use the HTTPClientConnection 212 at regular intervals of 10 minutes to obtain the latest stock quote data from the News Agency Server 320. The data is read in from the News Agency server using the HTTPClientConnection and Format and is decoded into the instance of TableContent. The QuoteHistoryCapture uses the SQLClientConnection 222 to load the data held in TableContent 220 into the database 330, so that a record of stock price movements is maintained.

Reference will now be made to Figure 23. The user selects via the browser's GUI the 'live stock quotes' hyperlink of the Stock Quote Home Page and causes the web browser 310 to issue a request for the 'live stock quote service' to the HTTPServant 202. The HTTPServer processes the service request which contains a request for the ResourceRegistry for an appropriate Resource which can provide the stock quotes. The request may be made using an ActorDiscoveryTemplate with the Role specified as 'Client' and the discriminator specified as "stockquotes" or a discriminator specific to the News Agency server. The QuoteHistoryCapture 204 and associated Resources meanwhile remain active in the background to keep the database server 330 updated.

Reference will now be made to Figure 24. The request from the web page of the browser, processed by HTTPServant 202, creates a new AppContext 232 with Attributes 236 which will be used to manage the data to be received. The LiveQuoteService is returned from the ResourceRegistry in response to the web page request and instantiated, contained and activated within the HTTPServant 202.

During activation the LiveQuoteService requests access to the AppContext 234 of the HTTPServant, and requests a client-side Connection for accessing the News Agency server. As a result HTTPClientConnection is 244 returned by the Registry and is instantiated and connected to the

NewsAgency webserver 320. The HTTPConnection 244 resolves the type of data that will be received from the News Agency server and requests the necessary Format from the ResourceRegistry, in step S246. As a result, StockQuoteFormat 248 is instantiated, contained and activated within HTTPClientConnection. The StockQuoteFormat 248 creates a TableContent 250, to receive the stock data. The LiveQuoteService registers as a ContentChange listener with TableContent 250 and activation is completed causing control to pass back to the request of the HTTPServant.

HTTPServant next requests that the LiveQuoteService perform its default operation, which is to download information from a news agency through the HTTPConnection 244. Consequently, the LiveQuoteService uses the HTTPConnection and StockQuoteFormat to receive data and decode it, in step S252, into TableContent 250. The LiveQuoteService 238 is registered with TableContent 250 and is therefore notified that TableContent has received data. The LiveQuoteService responds to the notification by creating a webpage to display the received stock data to the user. The LiveQuoteService requests DocumentContent 254 from the Registry, instantiates it and contains it. The TableContent data 250 is placed within the DocumentContent 254 by the LiveQuoteService. The DocumentContent 254 is then placed into the AppContext 234 provided by the HTTPServant 202. The LiveQuoteService also invokes the setExternalFormat of the DocumentContent to specify that it be encoded during externalisation into a HTML format.

The LiveQuoteService 238 completes its default operation, issues a PerformedOperation and is disposed of by HTTPServant 202. The HTTPServant 202 receives notification from its AppContext 234 that new Content has been added, i.e. DocumentContent 254, and requests from the ResourceRegistry, in step S260, an HTTPConnection which may be used to manage the transfer of the data in the DocumentContent 254 to the web browser 310. The HTTPConnection invokes the getExternalFormat of the Content data to determine the desired external encoding, and having done so requests an HTMLFormat from the Registry. The DocumentContent data is then read out through the Connection to be displayed on the web page of the browser 310. The HTMLFormat and HTTPConnection are not shown in Figure 24.

Reference will next be made to Figure 25. The user next decides that they would like to view the Stock Quote History, which is another option provided on the Stock Quotes homepage displayed on the browser. The user selects the option causing a request to be sent to the HTTPServant 202. The HTTPServant receives the request which contains a request to the Registry for a Resource called HistoricalQuoteService which presents stock price information. The HistoricalQuoteService 268 is accordingly returned by the Registry, instantiated, contained within HTTPServant 202 and activated. During activation the HistoricalQuoteService requests access to the HTTPServant's AppContext 234. The HistoricalQuoteService also requests from the ResourceRegistry a client-side connection for accessing databases through SQL. The SQLClientConnection 274 is then instantiated contained and activated within HistoricalQuoteService 268. During activation the SQLClientConnection requests, in step S276, and instantiates the SQLFormat 278 it requires to decode the data from the database, and a new TableContent 280. The SQLFormat is used by the SQLClientConnection 274 to decode the data into an internal TableContent 280. Once the data has been read in and decoded into the internal representation of TableContent 280, the SQLCLientConnection 274 and SQLFormat 278 are deactivated by the HistoricalQuoteService. The HistoricalQuoteService 268 next requests an AppContext 282 with Attributes 284, into which it puts the new TableContent 280. The HistoricalQuoteService 268 next requests from the ResourceRegistry an Actor that can generate graphs from data held in TableContents.

Reference will next be made to Figure 25. Class information for GraphGenerator 288 is obtained via the ResourceRegistry in response to the request from HistoricalQuoteService and an instance of the appropriate Actor instantiated, contained and activated. The HistoricalQuoteService 268 instructs the GraphGenerator 288 to perform its 'generate' operation causing it to create a DrawingContent 290 representing a graph of the movements of stock prices historically recorded in the database. The GraphGenerator 288 places the DrawingContent 290 into the AppContext 282 in place of the TableContent 280, in step S292, and completes its operation issuing a PerformedOperation to Attributes 284 of AppContext 282. The

HistoricalQuoteService 268 receives the notification and deactivates the GraphGenerator. HistoricalQuoteService 268 then creates a DocumentContent 294, consisting of the TableContent 280 and the DrawingContent 294, in S295, and places this, in step S296, into the AppContext 234 that it received from its container, HTTPServant 202. The HistoricalQuoteService 268 then completes it operation, issues a PerformedOperation to Attributes 236 of AppContext and is deactivated by HTTPServant 202. The HTTPServant 202 receives notification from the AppContext 234 that new Content has been added and so requests from the ResourceRegistry, in step S297, an HTMLFormat for encoding DocumentContent 294 in HTML suitable for viewing by the web browser 310. HTMLFormat 298 is instantiated and contained within HTTPServant 202 and used to pass the data to the web browser for viewing by the user in step S299.

Discussion

From the description above, it can be seen that the preferred system allows an application to be constructed at runtime that has a minimal memory footprint, that is it requires little memory and storage space which allows it to be deployed in devices that have limited memory capacity. In addition, the componentized nature of applications based on the invention means that the total size of those applications is considerably less than applications constructed by conventional means. Only those components that are needed are instantiated and take up room in the memory. This allows the deployment of the preferred system on existing "big systems" such as personal computers, high performance workstations, and server machines; and also smaller devices such as Personal Digital Assistants ("PDAs"), the "smart" mobile phones, consumer electronics such as games consoles and other entertainment systems, and embedded systems. It is also conceivable that the invention could be implemented as low-level control software to support dynamically adaptive operating systems and other control systems.

Componentized applications of the type that the preferred system makes possible increase competition in the software market, since only the components themselves need to be developed and marketed. The prices of the individual components would, needless-to-say, be much less than that of a whole application. Furthermore, a customer may buy only those components that they need.

The examples given in this specification refer to the use of the preferred system in conjunction with a core engine for a client-side applications or as an extension to existing client-side applications. Client-side applications are those programmes that execute on a user's machine and are controlled directly by users, such as Web and email servers. The preferred system may also be used with server applications, or for controlling and supporting middleware applications. Server applications accept requests from clients, perform some procedure such as reading files or processing information, and finally send the results back to the requesting client. Middleware applications provide additional functionality between clients and servers, and between multiple servers.

The preferred system may be used as the basis of a server product line, supporting the ability to incorporate new types of data, protocols, services, middleware, and client support services into a server-side environment. This system enables the integration of existing servers through middleware application components and to be expanded to support the functionality provided by those existing servers.

The ResourceRegistry and corresponding interface may be tailored to suit different software management and distribution policies. For example, a client side application suite may use a built-in implementation in which management of the installed Resources is performed through the application suite itself. An organization with a large number of users could use a stand-alone registry implementation that is accessed from a client-side application suite running on each user's machine.

While the preferred system is written in Java and is comprised of the APIs mentioned above, it is not a requirement that the invention is implemented as such, and it will be appreciated that the preferred system could be implemented in other programming or definition languages such as the Common Object Request Broker Architecture ("CORBA") Interface Definition Language ("IDL") and Microsoft's Component Object Model ("COM") and Distributed Component Object Model ("DCOM").

CLAIMS

1. A software application comprising:

a plurality of software modules each implementing a substantially different part of the application's functionality, that part being small in relation to the overall size of the application;

one or more instances of a data representation, being the data the application is manipulating;

one or more instances of an application configuration representation that contains information that the software modules may manipulate and that contains zero or one instances of the data representation; and

a containing environment in which the software modules, data instances and application configuration instances may be integrated into an application;

wherein the application is structured by encapsulating the software modules which implement the functionality used by another software module within that other software module.

2. The software application of claim 1, in which the data upon which the software modules are to act is accessed by the software module through the instance of the application attributes representation that contains that data.

3. The software application of claim 2, in which software modules organize themselves and communicate with one another by means of the information contained in the application attributes representation.

4. A method of classifying software modules, each of which implements a substantially different part of an application's functionality, that part being small in relation to the overall size of the application, according to their function so that they may be integrated into a coherent software application, the method comprising the steps of:

a) identifying the different types of functions that software modules implement within the application; and

b) specifying a set of methods that each software module of that type must implement to be integrated into the application.

5. The method of claim 4, in which the software modules are classified into four types, these being modules that implement functions for manipulating data that are available to the user, modules that decode and encode data between data encodings, and modules that manage the reading in of data and the writing out of the data from the application.

6. A system for synchronizing the operation of a plurality of different software modules, each implementing a substantially different part of an application's functionality, comprising zero or one instance of data for the software modules to manipulate and a representation of the attributes relating to the data on the application;

wherein the software modules are arranged to manipulate the instance of data, to modify the attributes to reflect changes in the data, and to record operations they have performed or changes in their own characteristics; and

wherein the software modules are arranged to monitor the attributes to be informed of the operations of other software modules.

7. A method of constructing a software application, comprising the steps of:

providing a first software module for receiving requests for functionality from a user, or another software application or software module;

providing a Registry in which software modules from which an application may be constructed may be stored;

providing a scheme of classifying software modules according to their function so that they may be identified and referenced;

installing information about software modules from which an application is to be constructed in the Registry; and

using the Registry at run time in response to requests for functionality from a user, from another software application or from a software module to obtain software modules which provide the requested functionality; and linking the software modules obtained from the Registry with a plurality of other software modules to form an application.

8. The method of claim 7, further comprising the steps of:

providing a representation of information describing a software module from which the software module itself may be obtained; and

installing the representation of information in the Registry.

129

The Patent Office
PATENTS • DESIGNS • COPYRIGHT • TRADE MARKS

INVESTOR IN PEOPLE

Application No:	GB 0031821.2	**Examiner:**	Geoff Western
Claims searched:	6	**Date of search:**	25 June 2001

Patents Act 1977
Search Report under Section 17

Databases searched:

UK Patent Office collections, including GB, EP, WO & US patent specifications, in:

UK Cl (Ed.S): G4A APX

Int Cl (Ed.7): G06F 9/44 9/46 9/52

Other: Dr Dobb's Journal. Online: JAPIO, EPODOC, WPI, Internet

Documents considered to be relevant:

Category	Identity of document and relevant passage	Relevant to claims
X,E	GB 2353612 A (MITEL) N.b. page 3	6
X	GB 2340265 A (SONY) N.b. pages 3-10	6
X	EP 0315493 A2 (VISYSTEMS) N.b. page 4 lines 11-35	6
X	US 6154763 A (THIJSSEN) N.b. columns 1-5	6
X	Dr Dobb's Journal, April 1997, CD-ROM version, "Java Beans • and the New Event Model", Giguere E.	6

X	Document indicating lack of novelty or inventive step	A	Document indicating technological background and/or state of the art.
Y	Document indicating lack of inventive step if combined with one or more other documents of same category.	P	Document published on or after the declared priority date but before the filing date of this invention.
		E	Patent document published on or after, but with priority date earlier than, the filing date of this application.
&	Member of the same patent family		